KRACKER JAX

REVEALED VOL II

TABLE OF CONTENTS

INTRODUCTION

Welcome to Kracker Jax Revealed Vol II. We at Kracker Jax want to thank you for your purchase and let you know that we do appreciate your support of our products. As a lot of you know, we polled many of the purchasers of Kracker Jax Vol I. Kracker Jax Vol II is the result of that response.

First of all, we will assume that you have purchased Kracker Jax Vol I and have performed many of the procedures in that Vol. The format of Vol II will remain similar, and we will continue with the cookbook approach. Please understand that we cannot be responsible for the machine language training that must be done before you can thoroughly understand the procedures and principles set forth in this manual. You don't have to be a fluent M/L programmer, but you must have a cursory knowledge of M/L and a strong natural curiosity. Don't expect (as some beginners do) a generic method of deprotection. It just does't exist. We can and will give you hints, tips, and techniques that can be applied to other programs, even if they are a completely different protection type than discussed in this manual. Finally, some protection schemes are based on the fact that no standard or nibble copier on the market can duplicate the program data. This protection becomes even harder to back up. No longer are we dealing with a sector or track of special protection, every byte on the disk becomes protected. These programs must be either broken from memory or a special copier developed to duplicate that program's format. Both of these methods are far too complicated to discuss within this manual. As you become more and more proficient at the patch method, the memory break method will become obvious. Writing copiers is in the realm of DOS experts that have a complete knowledge of M/L. Leave the special copiers to them.

KJ Revealed Vol II contains five new scheme types which we will call Type F, G, H, I and Z. Also, as promised in KJ Vol I, we have updated Type C. Two new entries have been added to Type E, to example programs that have no directory files. We will show you how to write parameters for these titles. The rest of the titles will be put into what we will call Type Z, a mixed bag. These titles all have one thing in common. They all have special drive routines written for them. Mastering these will give you a bag of tricks to pull from when the going gets rough. Lastly, we will be giving you our RapidLok(tm) copier system. This protection scheme was one of the most effective on the market for about two years. Even expert breakers pulled their hair out over this one. Because of their extremely difficult to copy format, we will not be showing you how to break these titles, but how the protection works and how to copy them.

Also included in this package is Hesmon(tm). We feel once you get used to using it, it will become an addiction. It's many features make it a joy to use. We will be using it whenever possible. For those situations that make Hesmon(tm) unusable, we have once again included our utility disk. Several of the utilities have been updated and a few more added. Again, since Kracker Jax Vol I is so important to understand, we will assume you have purchased it, and also have the reset button that was included. If you do not, we can supply you with one at a very reasonable cost (contact us).

For those of you who are serious protection breaking hobbyists, may we suggest a few additions to your breaking tools. First of all a three or four slot expansion board with a reset button can be an invaluable tool. Your

Hesmon(tm) can be flipped in and out at will. Super Snapshot(tm) is being used more and more by us in our breaking procedures. The monitor does NOT corrupt memory. Many programs are using protection that is located in screen memory. Super Snapshot's(tm) monitor can even execute code trapped in screen memory. In our opinion it is the most powerful utility of it's kind on the market. Finally, may we suggest our Hackers Utility Kit. This software was developed first of all to fit our demanding needs. We feel once you've used this collection of tools, you will wonder how you got along without them in the past. More info can be found in the back of this manual.

Lastly, for those of you who don't have many of the programs we have worked on in this manual, we have an answer. Rent A Disk located in West Virginia has consented to stock multiple copies of these titles. They are a mail order Disk Rental House and can fill your needs very nicely. Please understand that neither Rent A Disk or Kracker Jax advocate piracy in any form. We suggest this company only for educational purposes, and if you do work on a title you happen to like, they will be glad to sell it to you on a discount basis. Please do NOT write to their diskettes and please do NOT retain a copy of the program after breaking it. More details on Rent A Disk can be found in the back of this manual. Well enough of this. Let's get on to the breaking.

INTRODUCTION: PROTECTION SCHEME TYPE A
(Continued from KJ Revealed Vol I)

Owners of the 1541 disk drive may not realize it, but everytime they boot their favorite program and it bangs the disk drive head, that program is using this form of protection. It is common knowledge among experienced users, that this form of copy protection is hazardous to the health of the 1541 drive. Let's face it, would YOU write a program that purposely banged YOUR disk drive write head against it's end stop? This protection is still being used by many software publishers, knowing full well that the drive knock is probably the major source of disalignment problems with the 1541 disk drive. We at Kracker Jax can't see any purpose in the continuance of this form of protection. The user (and anyone else) can back up his software with almost ANY nibble utility on the market. The problem is, the backup is also protected and will bang the drive as well. It is this protection type that we especially urge you to learn to break, just so you can preserve the alignment of your disk drive.

The operation of this scheme is simple. The programmer writes a routine in the program (generaly in the boot) to seek out a non-standard sector on the disk. If that non-standard sector is found, the drive will usually bang, and the program will continue operations. If not, the program will cease to operate or "crash". These non-stanadard sectors are generally write errors, and are documented in your 1541-1571 drive manual. The most commonly used are the following:

20: Block header not found : drive banger

21: Sync character not found : sector not formatted properly, drive banger

22: Data block not present : drive banger

23: Checksum error in data : very common, drive banger

26: Attempt to write with write protect on : some programs check for the write protect, no drive bang

27: Checksum error in header : drive banger

29: Disk ID mismatch : whole track formated with wrong ID characters, no drive bang

Many of the programs using this scheme are checking the protection with simple drive commands and the kernal routines in the computer ROM. Keep in mind that this check can be done with BASIC programming as well as machine language. Once understood, most are fairly easy to deprotect.

Most of the time the programmer will check for the bad sector with a block read. It will look something like this: U1: aa bb cc dd (machine language) or B-R: aa bb cc dd (BASIC). The aa denotes channel, bb denotes drive number, cc denote track, and dd denotes sector. A character or two is then returned from the drive, and a comparison is made. If the comparison is satisfactory, the program continues operation. If not, the program flow is ended or set in an endless loop. Our task will be to either give the program the proper characters, or to short circuit the program flow around the protection check.

Kracker Jax Revealed Book one dealt with this scheme in four different programs. We have included this one title because this exact protection is a little tricky and has been found on quite a few programs.

Before starting to work on any of the following programs, please do a disk log, an error scan, noting all write errors, and use the C-64 Fast Copier to make a backup which will remove all errors. Place a write protect on the original disk.

< < < PROGRAM: ROGUE TROOPER$^{(tm)}$ <> PUBLISHER: UXB$^{(tm)}$ > > >

Procedure:

Loading the original produces a drive rattle. An error scan shows massive write errors on the original. A backup made with the C-64 Fast Copier produces a non working copy. Before starting to work on this program, do a disk log and an error scan to determine error type and location.

Working with your backup:

1/ Let's start by plugging Hesmon$^{(tm)}$ in the cartridge port and powering on. Insert your backup in the drive and load the boot file < LOAD"UXB",8,1 > . From the disk log we can determine that this file begins at memory location $032C. Start disassembly at $032C < D 032C > . Cursor down through the code. This code opens channels to the drive and loads a one character file name at $035B. If you Interpret memory at $035B < I 035B > you'll find the file name X. After the load, a jump to $08B0 is taken.

2/ Load the X file into memory < LOAD"X",8,1 > . Begin disassembly at $08B0. The following is an explanation of the program flow.

```
D $08B0 : JSR 081E
D $081E : $081E-$0841 opens an error channel to the drive < I E260 > and
          does a JSR back because the JUMP to $FFC0 is a kernal routine
          and always ends with a JSR.
D $08B3 : JSR 0844
D $0844 : Sends a U1 (Block Read) command to the drive from an
          encrypted form. The code from $084E-$085D decrypts and sends the
          U1.
D $0868 : JSR FFA5 : Imputs a byte from the serial port.
D $0872 : CMP 081A ($32 or the 2 in a 23 error).
D $0874 : BNE crash.
D $0877 : JSR FFA5 : Imputs a byte from the serial port.
D $087A : CMP 081D ($33 or the 3 in a 23 error).
D $087C : BQE to a JSR which closes channels and RTS back to $08B6.
          Otherwise the program flow falls through to a crash.
```

3/ There are many ways to break this title. Three will be given. Make all your changes using the Memory command and then resave the file to the backup as < S "@0:X" 08 0801 0977 > .

a. Place 3 NOPs at $08B3 over the JSR to $0844. This will cause the program to not even check protection.

b. Place a 30 at $081A and at $081D. This will allow the drive to send back a OK condition and pass protection because we will now be comparing to NO error.

c. Place a 60 (RTS) at $0844 which will cause the routine that checks protection to be short circuited.

4/ When your changes have been made, this title may be file copied.

< < < INTRODUCTION: PROTECTION SCHEME TYPE C > > >

(Continued From KJ Revealed Vol 1)

This protection scheme employs the use of a "Fat Track" to prevent the user from making his backup. To make matters worse, the Fat Track is placed on the outer (36-40) tracks.

Many of the examples covered in this manual work aproximatly the same. The following general loading procedure is taken with each.

1/ The boot is loaded and autostarts the program.

2/ A fast loader is set up and activated.

3/ The logo screen is loaded in and activated.

4/ The protection routine is decrypted.

5/ The files pertaining to the program are loaded in. These are generally encrypted.

6/ The protection is checked, which if satisfied ,places a numeric value ($FF) in the disk drive's memory.

7/ The value ($FF) is checked using a memory read.

8/ The value ($FF) is used as a part of a decryption routine to decrypt the main program. Proper decryption takes place ONLY if the correct value is returned.

9/ The code then jumps to the start of the program.

Four examples using this scheme were discussed in Kracker Jax Vol I. We must assume that you have mastered the techniques used to defeat those titles. Since that time, many titles have been released using Fat Tracks. Some were relatively simple to break and others were quite difficult. Some protection programmers have been checking not only for the Fat Track but also to see if either thier computer OR drive code had been tampered with. This was done by checksumming. If any sign of tampering was evident, the program refused to run - even if the break code was technically sound. If you have applied the methods in Kracker Jax Vol I to a similar protection, and it refused to work, you can assume they caught you in their code. We are going to give you examples of how to defeat the drive code, computer code, and the checksumming. Be advised, these examples show tricks and techniques that can be used again on other schemes. Breaking protection involves thought and ingenuity.

< < < PROGRAM: TITANIC(tm) <> PUBLISHER: ACTIVISION(tm) > > >

Procedure:

Loading the original disk produces a rattle free load. and an error scan shows no standard errors. A backup made with the C-64 Fast Copier produces a non working copy. A backup made with a nibbler produces the same non working backup. Before starting to work on this program, please make a (non working) backup of the original, and a disk log to log the file addresses.

Working with your backup:

1/ Let's start by plugging Hesmon(tm) in the cartridge port and loading the boot < L "*" 08 > . Checking with the disk log, start disassembly of code at $02D7 <D 02D7> and cursor down through the code. The code from $02EE to $0301 opens a channel for loading, sets the file name " 1985 ", loads that file in and Jumps to $4635. We can load that file in ourselves and inspect it.

2/ Cursor down to a clear spot and load the 1985 file as < L " 1985*" 08 > . Be sure to use two spaces before the 1985 file name. The disk log shows this file ranges from 4400-46D8. Look at the file in ASCII by using the Interpret command <I 4400> and cursor down through memory. Take note of what it looks like, because we will be looking again later. Let's start disassembly at the Jump to $4635 <D 4635> . Cursor down through the code and note code from $4657 to $4668. Values are being set for the decrypter at $466F to $4690 (see Kracker Jax Revealed Vol I for more details). We want to execute the decrypter and stop the execution after the decrytion takes place. To do this we must place a 00 (Break Instruction) at $4690. Use the Memory command to make your change <M 4690> and change the 60 to a 00 and hit return. Now we can decrypt the code by executing at $4657. Use the GO command <G 4657> .

3/ When the monitor breaks, use the Interpret command again starting at $4400 <I 4400> and cursor down through memory again. This time note the Block-Execute at $4571. This command opens channel 2, addresses drive 0, and sends the code at track 3 sector 0 to the RAM of the disk drive ($0400 in this case) and executes the code in the drive. This code is the protection check routine. While in the Interpret mode, also note the U1 (Block-Read) of the same Track 3/Sector 0. This block read is used to checksum the drive code to check for tampering. Checksums throughout the computer code also check strategic areas of the computer code for tampering. If changes in the original code are found, the program will not run even if the break is correct. Here's a trick to break the drive code and still keep the checksums intact.

4/ Turn the computer off and back on again to clear memory. X to BASIC <X> and from the Utility Disk, load the Block Read file < LOAD"BLOCK READ",8 > . When the ready prompt comes up. LIST the file and on line 10 set the TRack variable to 03 and the SEctor variable to 00. Hit RETURN to lock your changes in and relist the file to check your changes. This utility will Block Read Track 3/Sector 0 and send the code to $C000 in the computer where we can inspect it. Place the backup in the drive and start the Block Read by Typing RUN and hitting return. The drive will spin and in about 30 seconds, the ready prompt will appear. Return to the monitor by hitting Run/Stop-Restore. Disassemble code at

$C000 <D C000> . Cursor down through the code. The code from $C000-$C011 is the decryptor and will have to be executed before we can inspect the drive code. You'll see that it is set to decrypt this code in the $0400 Buffer in the drive and must be readdressed to decrypt at $C000. Using the Memory Command. change the 04 at $C006,$C009,$C00C,and $C00F to C0. Now Disassemble starting at $C000 again and check the decrypter again. It should now be set up to decrpyt code in the $C000 buffer.

5/ Let's execute the decrypter and inspect code. Type <G C001>, and when the monitor breaks, Disassemble code at $C000 <D C000> and cursor down through the code. The code from $C012-$C04C checks Track 35, bumps the head a half track and if the check is satisfactory, stores a 0 in $0009. The Instruction at $C04D loads the accumulator with the value in $0009. Next, if that value is not a 0, the code branches around the next two instuctions. These are the keys to the protection. The value of $FF is stored at $01FF in the drive memory. Later a Memory Read in the computer code will check for the $FF and if it is in place at $01FF, the protection check will be passed. Our job now is to force this routine to pass even if the protection isn't in place. One way would be to place two NOPs ($EA) at $C050 to erase the BNE C057. This would force the code to fall through and store the $FF byte even if protection wasn't passed. This would work, but the checksum would catch us. Here's a trick to force the code to fall through and still pass the checksum.

6/ Because the key to this break is the BNE command at $C050, let's flip those bytes and see what instruction comes up. Use the Memory command to change the D0 05 at $C050 to 05 D0 <M C050>. Disassemble $C050 again <D C050>. The BNE instruction has now become an ORA D0. This has effectively negated the BNE because this instrucion is essentially worthless and performs no task that is actually used. The checksum will also pass because we haven't actually changed any bytes, only their position. Let's prepare to make our changes to the disk. Turn off the computer and remove Hesmon$^{(tm)}$.

7/ From the utility disk, load and run the Disk Doctor. Place the backup in the drive and using the b <b> command read in Track 3/Sector 0. At position $50 (remember $C050), 80 in decimal you'll find the two bytes that we need to flip. These are $D4 and $01. Remember, these are the bytes in their encrypted form. Change these to $01, $D4. You may use the @ key and the decimal values. Starting at position 80, change two bytes to 01, 212. Hit r <r> to rewrite the block and y <y> for yes. This title is now broken from protection, and may be fast copied. Because of the Block Execute to Track 3/Sector 0, you may not file copy this title. The drive code, even though broken, must be in place on the disk.

< < < PROGRAM: ROCKY HORROR SHOW(tm) <> PUBLISHER : ACTIVISION(tm) > > >

Procedure:

Loading the original disk produces a rattle free load. and an error scan shows no standard errors. A backup made with the C-64 Fast Copier produces a non working copy. A backup made with a nibbler produces the same non working backup. Before starting to work on this program, please make a (non working) backup of the original, and a disk log to log the file addresses.

This break method is presented to add a trick to your arsenal. If it is confusing at first, a little studying of the code will make the break clear. Print-outs of any confusing code may also help to make things clear. We assume you have performed at least one Activision(tm) break from K J Revealed Vol 1. Please follow instructions closely.

Working with your backup:

1/ We will start by filling the BAM with zeros so the drive will be fooled into believing our backup disk is full. This way we can scratch and then save a file back to the disk without overwriting any program code that isn't allocated in the BAM. Use this trick whenever you suspect any hidden files not in the directory.

Load Disk Doctor from the Utility Disk. Place the backup in the drive and go to Track 18/Sector 0 using - command. This is the BAM sector. Using the @ key, fill position 4 through 71 with zeros (@). Skip over 72 to 75 which is the directory track and fill 76 through 143 with zeros (@). When finished, rewrite the changes to the disk by hitting <r> for rewrite and <y> for yes.

2/ With Hesmon(tm) in the cartridge port, load the boot < L "*" 08 > . Checking with the disk log, start disassembly of code at $02D7 < D 02D7 > and cursor down through the code. The code from $02E8 to $0301 opens a channel for loading, sets the file name " 1985 ", loads that file in and Jumps to $135A. We can load that file in ourselves and inspect it.

3/ Cursor down to a clear spot and load the 1985 file as < L " 1985*" 08 > . Be sure to use two spaces before the 1985 file name. The disk log shows this file ranges from 1000-143F. Look at the file in ASCII by using the Interpret command < I 1000 > and cursor down through memory. Take note of what it looks like, because we will be looking again later. Let's start disassembly at the Jump to $135A < D 135A > . Cursor down through the code and note the decrypter code from $139B to $13BC. We want to execute the decrypter and stop the execution after the decrytion takes place. To do this we must place a 00 (Break) at $1398. Use the Memory command to make your change <M 1398> and change the 4C to a 00 and hit return. Now we can decrypt the code by executing at $137D. Use the GO command < G 137D > . After the monitor breaks, use the Interpret command to examine the code from $1000-$143F again < I 1000 > . You'll find it to be quite different now and you should be able to see quite a few commands in ASCII. Finally use the Memory command to change 00 we placed at $1398 back to a 4C < M 1398 > .

4/ Let's trace the code starting at $135A commenting the code pertaining to the protection check.

```
$135A-$1394     : Sets up the decrytion values.
$1395 JSR 139B : Executes decryption of   1985   file.
$1398 JMP 13BD : Jump around decrypter already executed.
$13BD JSR 1184 : JSR to protection check.
$1184 JSR 1206 : Sets up for protection check.
$1187 JSR 118E : checks drive memory for a value of $FF at $01FF. EORS that
                 value with an $FF which produces a Zero (0). Places that
                 zero at $1294. Later the value at $1294 is used in the
                 program decryption.
$118A JSR 1269
$118D RTS
$13C0 JSR 1116 : Continue on.
```

5/ This protection would be simple to deprotect if it weren't for the checksums used throughout the code. Every strategic point has been checked and if we are caught tampering with the code, the program won't work, even if the break is sound. We need to trick the checksums. Testing in various spots has uncovered an area that is not checksumed. The decrypter routine is not checked and if moved, will provide us with a work area to place our code in and short circuit the protection check. Let's begin here.

6/ Reload the 1985 file to provide fresh undecrypted code < L " 198*" 08 > . First let's move the decrypter to the outside bounds of this file. Since the file ends at $143F we can move it to $1440. Use the Transfer command < T 139B 13BC 1440 > . Disassemble code at $1440 <D 1440> and cursor down through the moved decrypter. You'll find the last byte, a $60 at $1461. This will become the new end address of this file.

7/ Now that the decrypter has been moved, lets prepare the workspace. Fill the area from $1395-$13BC with NOPs < F 1395 13BC EA > . Now let's use the assembler in Hesmon$^{(tm)}$ to rewrite the code in our workspot. A printout of the prior code to compare with our changes should make the reasons for our changes clear. We can start writing our code a $139A. Start by using the assemble command < A 139A > . Here's the code to write.

```
A 139A JSR 1440 : Decrypt code from new decrypter location.
       JSR 1206 : Set up for protection. -------* code from here
       LDA #$00 : Substitutes for protection-----* to here will replace
       STA 1293 : check at $118E----------------* the JSR 1184 at
       JSR 1269 : ------------------------------* $13BD.
       JMP 13C0 : Jump around JSR 1184 at $13BD, which is no longer needed
```

When done, hit return a few times to a clear spot and Disassemble code and check to make sure the changes are correct < D 139A >. If all is well, all that's left is to scratch the old file and save the new. X to BASIC <X> and scratch the 1985 file. < OPEN15,8,15,"S0: 1985   " > . Be sure to use two spaces before 1985 and three spaces after. When done, hit Run/Stop-Restore to re-enter the monitor and save the new 1985 file. Our new start/end addresses are $1000-$1461+1. < S " 1985   " 08 1000 1462 > You're backup is now completely broken and may be fast copied. You can't file copy this title because of the various Block-Executes used in the loader for the fast load routine as well as protection checks. These Block-Executes access code not allocated by directory files.

< < < PROGRAM: TRIO(tm) <> PUBLISHER: SOFTSYNC(tm) > > >

Procedure:

Loading the original disk produces a rattle free load, and an error scan shows no standard errors. A backup made with the C-64 Fast Copier produces a non working copy. A backup made with a nibbler produces the same non working backup. Before starting to work on this program, please make a (non working) backup of the original, and a disk log to log the file addresses. Please note the XEMAG 2.0 in the directory header. This is the signal to you of Fat-track protection.

Working with your backup:

1/ Let's start by plugging Hesmon(tm) in the cartridge port and loading the boot < L "*" 08 > . Checking with the disk log, start disassembly of code at $02A7 <D 02A7> and cursor down through the code. The code from $02C3 to $02C9 loads in a file with 7 characters in it's name. Interpret memor·· at $02A7 <I 02A7> to see that file name. You'll find a name using a combination of regular and reverse characters. Again disassemble memory at $02A7 <D 02A7> and cursor down through the code. At $02F7 you'll find a jump to $A483 which causes BASIC to execute.

2/ Power off and on again. When the monitor appears, <X> to BASIC and load and list the directory < LOAD "$",8 > . Near the end, you'll find the file with regular and reverse characters. Load that file directly from the directory with a <,8:> . When the READY prompt comes up, cursor down to a clear spot and list that file. Examination of this file shows that it loads and runs the TRIO CALC, TRIO WORD, OR TRIO FILE depending on the menu choice picked by the user.

3/ Again cursor down to a clear spot and load TRIO FILE < LOAD "TRIO FILE",8: > . List out this file for examination. This program loads the file TRIO3, does a sys 32768 ($8000) to it, comes back, and reads drive memory at $01FF and compares the value there to a (2 up arrow 8-1) which is a decimal 255 or $FF. If the value is not equal to an $FF, a NEW occurs which crashes the program. If it is equal to $FF then the program falls through to a GOTO 70. (You'll find similar programming in the TRIO WORD and TRIO CALC files.)

4/ Because the file TRIO3 resides at $8000, which is where our Hesmon(tm) cartridge resides, we must use a different monitor. Turn off the computer and pull the Hesmon(tm). From the Utility Disk, load the $2000 monitor < LOAD "8192",8,1 > . WHEN THE READY prompt comes up, sys the monitor in <SYS 8192> . Load the TRIO3 file from the TRIO backup < L "TRIO3",08 > and start disassembly at $8000 <D 8000> . The code from $8000 to $8036 does a BLOCK EXECUTE to Track 35/Sector 10/. $8037 to $8062 MEMORY READS the drive at location $01FF and compares to an $FF. If the value is not equal to an $FF, then a branch to $8070 takes place. To see what happens, cursor to a clear spot and do a Go $8070 <G 8070> . When done, hit Run/Stop-Restore and again sys the monitor in with <SYS 8192> . Again disassemble code at $8000 and cursor down through the code. You'll find that if the comparison to $FF is satisfactory, the programming falls through to $808B, which is a JUMP to $FFC3. This is a KERNAL routine that when JUMPed tc, does a RTS which in this case returns the program flow back to the basic program (TRIO FILE in this case.).

5/ Turn the computer off, insert the Hesmon(tm), and power up again. X to BASIC <X> and from the Utility Disk, load the Block Read file < LOAD"BLOCK READ",8 > . When the ready prompt comes up. LIST the file and on line 10 set the TRack variable to 35 and the SEctor variable to 10. Hit RETURN to lock your changes in and relist the file to check your changes. This utility will now Block Read Track 35/Sector 10 and send the code to $C000 in the computer where we can inspect it. Place the backup in the drive and start the Block Read by Typing RUN and hitting RETURN. The drive will spin and in about 30 seconds, the READY prompt will appear. Return to the monitor by hitting Run/Stop-Restore. Disassemble code at $C000 <D C000> . Cursor down through the code. The code from $C000-$C010 is the decryptor and will have to be executed before we can inspect the drive code. You'll see that it is set to decrypt this code in the $0400 Buffer in the drive and must be readdressed to decrypt at $C000. Using the Memory Command, change the 04 at $C005 and $C00B to C0. Notice the ADC $08 at $C007. This instruction uses the value in the drive at location $08 to help decrypt this code. The location $08 is the track value last loaded into the Buffer at $0400. We know that this was track 35 (remember the BLOCK EXECUTE to Track 35/Sector 10). Let's change the instruction from a ADC $08 to a ADC #$23. We are now using the known value of $23 (decimal 35) and not using any values in drive memory. The bytes for this instruction change are $69,$23. Use the MEMORY command to make your changes at $C007 < M C007 >. Again disassemble memory at $C000 and cursor down through the code to check to see the changes are correct.

6/ Let's execute the decrypter and inspect code. Type <G C001>, and when the monitor breaks, dissasemble code at $C000 <D C000> and cursor down through the code. The code from $C011-$C043 checks Track 35, bumps the head a half track and if the check is satisfactory, stores a 0 in $0009. The instruction at $C044 loads the accumulator with the value in $0009. Next, if that value is not a 0, the code branches around the next two instuctions. These are the keys to the protection. The value of $FF is stored at $01FF in the drive memory. Later a Memory Read in the computer code will check for the $FF and if it is in place at $01FF, the protection check will be passed. Our job now is to force this routine to pass even if the protection isn't in place.

7/ One way to break this code is to write a simple routine to place an $FF in drive location $01FF and return to the programming that sent it in the first place. This is accomplished simply. Cursor down to a clear spot and go into the ASSEMBLY mode by typing <A C000> . Here's the code:

```
   A C000 LDA #$FF <RET>  (A9 FF)
   A C002 STA 01FF <RET>  (8D FF 01)
   A C005 RTS      <RET>  (60)
```

When done, cursor down to a clear spot and disassemble at $C000 <D C000> to see the bytes needed. You'll find the following six bytes: A9 FF 8D FF 01 60. You can use the hex to decimal converter in Hesmon(tm) to convert the bytes to decimal <$A9 RET, and so on>. You'll find that the following is the decimal equivalent: 169 255 141 255 01 96.

8/ From the Utility Disk, load and run the Disk Doctor. Place the backup in the drive and using the b <b> command read in Track 35/Sector 10. Starting at position $00, write in the six bytes. You may use the @ command to write them one at a time in Decimal (169 255 141 255 01 96). When the changes have been made, hit r <r> to rewrite the block and y <y> for yes. This title is now broken from protection, and may be fast copied. Because of the Block Execute to Track 35/Sector 10, you may not file copy this title. The drive code, even though broken, must be in place on the disk.

< < < PROGRAM: ALIENS$^{(tm)}$ <> PUBLISHER: ACTIVISION$^{(tm)}$ > > >

Procedure:

Loading the original produces a rattle-free load, and an error scanner shows no standard errors. A backup made with the C-64 Fast Copier produces a non-working copy. A nibbled backup produces the same non-working copy. Before starting to work on this program, please make a (non-working) backup of the original, and a disk log to log the file addresses.

1/ Turn off your computer and insert your reset button assembly (from KJ Revealed Vol I) into the cartridge port. Turn on the computer again. Load the $C000 monitor from your Utility Disk < LOAD"49152",8,1 >. At the completion of the load, type < SYS49152 > and hit < RETURN >. The monitor should be active now.

2/ With your backup in the drive and the monitor active, load the boot file < L"0:*",08 > . When the load is complete, disassemble memory at $02CB. You'll find a loader routine that loads in the "ACTIVISION INC." file and jumps to $8000.

3/ Load the "ACTIVISION INC." file into memory < L"A*",08 >. After the load, start disassembly of code at $8000 < D 8000 > . Also do an ASCII dump < I 8000 > to check for DOS commands. Examine the routines carefully. You will soon find a Block-Execute (B-E 2,0,18,7) drive command at $80DD. Further examination of the code reveals that the protection scheme is doing a lot of direct access to the serial port at $DD00.
The key to cracking this variation on Activision's$^{(tm)}$ standard protection scheme is to ignore this code because it has a rather involved loop that is a pain to follow and de-protect. With this code, the drive is where the action's at. Let's take a closer look at that Block-Execute code on track/sector 18/7.

4/ Reset the computer and load ALIENSLOADER from the Utility Disk < LOAD "ALIENSLOADER",8 > , < RUN > and follow the instructions. Reload the 49152 monitor and < SYS49152 > . In the drive, the code would be located at $0300. We will be using using $2300 (in the computer). Disassemble the code at $2300 < D 2300 > . The routine at $2322 - $234A, despite it's apparent complexity, does nothing more than load the code in track/sector's 18/7 - 18/11 into drive memory locations $0400 - $07FF. The ALIENSLOADER routine has conveniently loaded these for us already. The code, from $2400 - $27FF, is encrypted. A routine at $2356 does the decryption. We can modify the code to decrypt it for us by simply adding $2000 to the LDA and STA address references, i.e. $0400 becomes $2400, $0500 becomes $2500, etc..< A 2358 LDA $2400,Y etc.. > . Also put a break command at $237F < A 237F BRK > and run the code < G 2356 > .
NOW examine the code starting at $2400 < D 2400 > . Most of this code is the fast loader. Armed with the knowledge that Activision$^{(tm)}$ fat tracks start with track 35 ($23), we find a suspicious routine at $24D0 - $24F8. This is it, folks. This itty-bitty loop is the heart and soul of this protection scheme. It can be disabled easily with one byte change. Change the LDA operand byte at $24DE from $80 to $01 < A 24DD LDA #$01 >. Instead of READING the intended sector, the $01 byte tells the drive's DOS that the job was completed succesfully. This is exactly what you want it to do. The fringe benefit of this method is that the program loads about 8 seconds faster and you'll hear a

pleasant clicking noise when the protection scheme executes the code with your byte change (when the screen blanks).

5/ Re-encrypt the code using the same routine at $2356 < G 2356 > . Before we load up the sector editor to write the bytes back, let's look back at the decryption loop at $2356 < D 2356 > : it's exchanging bytes between $2400 / $2500 and $2600 / $2700. Our changed byte (now $54) is at $25DE, -not- at $24DE. It will be written to track/sector 18/9 at position $DE (222 decimal).

6/ Now reset the computer, re-insert the Utility Disk and reload the sector editor < LOAD"DISK D*",8 > . Insert your backup and < RUN > . Press the < B > key. Enter 18 < RETURN > and 9 < RETURN > to read in track/sector 18/9. Move the cursor to position 222 and press the @ key. Enter 84 and press < RETURN > . To write the modified sector, press < R and Y > .

7/ Reset and load the backup. It DOES load faster than the original, doesn't it?

< < < PROGRAM: TRANSFORMERS$^{(tm)}$ <> PUBLISHER: ACTIVISION$^{(tm)}$ > > >

Procedure:

Loading the original produces a rattle-free load, and an error scanner shows no standard errors. A backup made with the C-64 Fast Copier produces a non-working copy. A nibbled backup produces the same non-working copy. Before starting to work on this program, please make a (non-working) backup of the original, and a disk log to log the file addresses.

1/ Turn off the computer and insert your reset button assembly (from KJ Revealed Vol I) into the cartridge port. Turn on the computer again and load the $C000 monitor from your Utility Disk < LOAD"49152",8,1 > . At the completion of the load, type < SYS49152 > and hit < RETURN> . The monitor should be active now.

2/ With your backup in the drive and the monitor active, load the boot file < L"COP*",08 > . When the load is complete, disassemble memory at $02E0. You'll find a loader routine that loads in the " 1986 " file and jumps to $0506.

3/ Because the " 1986 " file loads into screen memory where we normally can't look at it, we must first change the load address to something more accessible. Reset the computer, insert the Utility Disk and load the sector editor < LOAD "DISK D*",8 > . Insert your backup disk and < RUN > . Go to track/sector 18/01 < B 18 RETURN 1 RETURN >. Cursor over to position 35. The first sector of "1986" is 17/01 ($11/01-hex). Jump to there < j > . Move to position 3, press the '@' key and change the byte $05 to 37 ($25) and press RETURN. This changes the load address to $2500. Write the sector back to disk < R Y > and reset your computer.

4/ Again insert the Utility Disk and load and activate the 49152 monitor. Load the " 1986 " file into memory < L" 19*",08 > . After the load, start disassembly of code at $2500 < D 2500 > . Also do an ASCII dump < I 2500 > to check for DOS commands. Examine the routines carefully. You will soon find a Block-Execute (B-E 2,0,1,1) drive command at $271E. Further examination of the code reveals that the protection scheme is doing a lot of direct access to the serial port at $DD00.

The key to cracking this variation on Activision's$^{(tm)}$ standard protection scheme is to ignore this code because it has a rather involved loop that is a pain to follow and de-protect. With this code, the drive is where the action's at. Let's take a closer look at that Block-Execute code on track/sector 1/1. (Before going on to step five, change the load address of the " 1986 " file back to $0500. Use the same procedure as outlined in step 3.

5/ Reset the computer and load TRANSLOADER from the Utility Disk < LOAD "TRANSLOADER" ,8 >, < RUN > and follow the instructions. Reload the 49152 monitor and < SYS49152 > . In the drive, the code would be located at $0300. We will be using using $2300 (in the computer). Disassemble the code at $2300 < D 2300 > . The routine at $2321 - $2349, despite it's apparent complexity, does nothing more than load the code in track/sector's 1/2 - 1/5 into drive memory locations $0400 - $07FF. The TRANSLOADER routine has conveniently loaded these for us already. The code, from $2400 - $27FF is encrypted. A routine at $2355 does the decryption. We can modify the code to decrypt it for us by simply adding $2000 to the LDA and STA address references, i.e. $0400 becomes $2400, $0500 becomes $2500, etc...< A 2357 LDA $2400,Y etc.. > . Also put a break command at $237E < A 237E BRK > and run the code < G 2355 > .

NOW examine the code starting at $2400 < D 2400 > . Most of this code is the fast loader. Armed with the knowledge that Activision(tm) fat tracks start with track 35 ($23), we find a suspicious routine at $24B4 - $250F. This is it, folks. This itty-bitty loop is the heart and soul of this protection scheme. It can be disabled easily with one byte change. Change the LDA operand byte at $24C2 from $80 to $01 < A 24C1 LDA #$01 > . Instead of reading the intended sector, the $01 byte tells the drive's DOS that the job was completed succesfully. This is exactly what you want it to do. The fringe benefit of this method is that the program loads about 8 seconds faster and you'll hear a pleasant clicking noise when the protection scheme executes the code with your byte change (when the title screen appears).

6/ Re-encrypt the code using the same routine at $2355 < G 2355 > . Before we load up the sector editor to write the bytes back, let's look back at the decryption loop at $2355 < D 2355 > : it's exchanging bytes between $2400 / $2500 and $2600 / $2700. Our changed byte (now $54) is at $25C2, -not- at $24C2. It will be written to track/sector 1/3 at position $C2 (194 decimal).

7/ Now reset the computer, re-insert the Utility Disk and reload the sector editor < LOAD"DISK ?*",8 > . Insert your backup and < RUN > . Press the <B> key. Enter 1 < RETURN > and 3 < RETURN > to read in track/sector 1/3. Move the cursor to position 194 and press the < @ > key. Enter 84 and press < RETURN > . To write the modified sector, press < R > and < Y > .

8/ Reset and load the backup. It DOES load faster than the original, doesn't it?

< < < INTRODUCTION : PROTECTION SCHEME TYPE E > > >

(Continued From KJ Revealed Vol I)

This protection scheme is, at this writing, one of the most effective, and prevelant methods of defeating today's nibble copiers. When you know what to look for, you'll find this scheme is being employed by many different software houses. I like to think of this protection as the big brother of the long sectors discussed in Section D.

This scheme can be recognized by the following similarities. When a disk error check is done, no write errors will be found on the original. When booted, no drive rattle will be encountered. The program cannot be backed up with either a fast copier or a nibbler. Usually, you will find data in the directory other than normal directory data. Most important: when tracing the program through it's loading process, you will generally run into a decryption routine and a sector or two of encrypted code. When this encryption is located, you can be sure it is hiding the protection check code.

Remember, I stated that a sector or two in memory will be encrypted, and that this area in memory surely contained the protection check. Well, one other thing needs to be mentioned. This is the fact that this encrypted memory starts out as garbled code, then decrypts into a protection check routine and finally after the protection check has been satisfied, is REPLACED with valid program code. This code, as previously stated is one or two sectors in length and can be found anywhere on the program disk. You'll find that the directory track (Track 18) is the most likely spot. In most cases, we can let the program insert the hidden code in it's proper place. Then a memory save and replacement over the encrypted code in the proper file (if there are directory files) will not only defeat protection but will totally remove the check for it.

Most of the programs protected with this scheme can be defeated with a simple memory save, but a few have had to have some of the code re-written by hand. This is relatively uncommon and cannot be explained in the scope of this manual. Experience will prove to be the best teacher.

Before starting to work on the following programs, please format a blank work disk, and have a (non-working) backup available. Please make sure you have a write protect on your original program disk, as you will be using it in the breaking process. Now let's get on to the specifics.

< < < PROGRAM: EXPRESS RAIDER(tm) <> PUBLISHER: DATAEAST(tm) > > >

Procedure:

Loading the original disk produces a rattle free load, and an error scan shows no standard errors. A backup made with the C-64 Fast Copier produces a non working copy, A backup made with a nibbler produces the same non working backup. Before starting to work on this title, make a fast-copier backup and a formated work disk. Because the only file on the directory of this title is the loader, special procedures will be required. You will need a reset button of some sort (provided in Revealed Vol 1).

Working with your backup

1/ With the reset switch in place, load the backup three or four times to get the feel of when the program stalls. When you have gotten the timing down, try to reset the computer just before that stall occurs. You will hear the head swing out if you are too late. We want to reset just before it does. After reset, from the Utility disk, load the $C000 monitor < LOAD "49152",8,1 > and after the load sys it in < SYS 49152 > .

2/ If you have performed the breaks in Section E from KJ Revealed Vol 1, you will remember that we are looking for a decrypter that hides the protection check. That decrypter ALWAYS begins with A0 00 A9. So we can search most of memory, flip out the BASIC Interpreter by changing memory location $0001 from a $37 to a $36 ($76 on the C-128) < M 0001 > . Now do a hunt for the key bytes in memory < H 0800 BFFF A0 00 A9 > . If you have reset out at the proper time, the following addresses will be returned: 84C0 8759 9629 9A4C . Start by disassembling $84C0 < D 84C0> and inspecting the code below that address. If the code is clean, it is not what we are looking for. Inspect all the returned addresses and look for programming that has code beneath it that does not disassemble properly (usually you'll find a lot of ?????) . You'll find that $9626 fits the bill exactly. Here you'll find the decrypter with about a sector of encrypted code beneath it.

3. Because of the no directory files problem, this break poses a slight inconvenience. We will have to search the disk for the proper place to lay down the break code. This type of loader uses a Track & Sector method of loading. You'll find that each page in memory occupies its own sector on the disk. Because the break code is between $9600 and $9800, we need to record the first 5 or 6 bytes from $9600 and $9700 to make it easier to locate these on the disk. (Remember these will be the first bytes in the sectors they occupy.) Using the Memory command, inspect and record the first few bytes in each: $9600= 96 4C E0 97 4C FB 97/$9700= 40 ED 84 99 01 99 74. Again disassemble memory at the decrypter and use the cursor key to scroll down through memory < D 9626 > . You must scroll down at least a full sector ($9726) and a bit more, until you see clean code again. At $9736 you'll find a JUMP t0 $9744 (4C 44 97). Record this information for later referance.

Working with your original:

4/ Power off and on again to clear memory. Load the original disk until the game has started up and again hit the reset button. From the Utility Disk, again load and activate the $C000 monitor as before. Start disassembly at $9626 < D 9626 > . You'll find new code has replaced the previous encrypted code. The key to breaking this type of protection is to replace the encrypted code with this new code. Disassemble again at $9626 and cursor down through memory. At $9736, you'll find the same three bytes as we recorded earlier: 4C 44 97. This tells us that the code from here on is the same as it was in the unrun and encrypted state. Place your formatted work disk in the drive and save the new code < S "BLOCK",08,9626,9738 >

Working with your backup:

5/ Our task now is to transfer the code in the BLOCK file to the backup disk in the proper location. Here's the procedure. Power off and on again. Load the Disk Dr from the Utility Disk and RUN it < LOAD "DISK DOCTOR",8,1 > . Using the - command from Disk Dr., search from Track 18/Sector 0 backwards one sector at a time. You'll be looking for the Sector that contains 96 4C E0 97 4C FB 97 as it's first seven bytes ($9600 in Memory) and 40 ED 84 99 01 99 74 as it's first seven bytes ($9700 in Memory). This search is time consuming but necessary. You will find that $9600-$96FF will be at Track 11/ Sector 8 and $9700-$97FF at Track 11/Sector 16. Thus the code must be placed at Track 11/Sector 8 Position $26 (38 in decimal) and continues on to Track 11/Sector 16 position $00 to end.

6/ Using Hesmon$^{(tm)}$, convert our BLOCK start and end addresses to decimal. $9626 = 38438 and $9736 = 38710. Power down and remove Hesmon$^{(tm)}$. Now let's begin creating the parameter that will lay down the saved code in the proper location on the backup for us. Follow these instructions precisely.

a. From the work disk load the BLOCK file < LOAD "BLOCK",8,1 > .
b. Type NEW and hit RETURN.
c. From the Utility Disk load the PARM TEMPLATE < LOAD "PARM TEMPLATE",8 > .
d. List out the template and inspect. Start the data maker by typing GOTO600
e. Hit RETURN to continue. Enter Start as 38438 and END as 38710.
f. Record the number of bytes for use later (273 bytes) and hit RETURN.
g. The datamaker will now PEEK memory where our BLOCK is stored and convert the bytes to data statements in decimal.
h. When the program ends, LIST again. Edit line 5 for the desired title.
i. List out line 100 and Edit :TR=11:SE=8:FB=38:NB=218
   Tr=TRack(11),SE=SEctor(8),FB=First Byte Position (38), NB=NUmber of bytes (218) <256-38=218>. Hit RETURN to lock in.
j. Type a 101 over the 100 in line 100 and Edit : Tr=11:SE=16:FB=00:NB=55
   Tr=TRack(11),SE=SEctor(16),FB=First Byte Position (00), NB=NUmber of bytes (55) <273-218=55>.Hit RETURN to lock in.
k. Save the new parm to the work disk < SAVE "TEST",8 > .

7/ Now run the parameter on the backup. Load the backup, and test it. You'll find that it doesn't work. Some titles require a little more work. Again with the reset switch in, load the original again, resetting just before the head swing. Again load the $C000 monitor and sys it in < SYS 49152 > . We need to find the routine that either does a JSR or a JMP to the protection routine at $9626. Again change the $0001 address to $36 and use the HUNT command to search for a JSR 9626 or a JMP 9626 < H 0800 BFFF 20 26 96 >, and < H 0800 BFFF 4C 26 96 >. You should get a 76A1 returned. Disassembly of $76A1 shows a JSR 9626. Occasionally you will have to change the JSR (20) to a JMP

(4C) or completely erase it with NOP's EA EA EA. As before, record the bytes at $7600 so we may find the sector containing this code on the disk. $7600 = 2F 8D 11 D0 8E 20 D0 8E. Again power down and on again and load the Disk Dr from the Utility disk. Search the first bytes of each sector untill you locate the desired pattern. We found it at Track 19/Sector 10. The 20 26 96 bytes are located at position $A1 (72 in decimal).

8/ Reload the TEST parameter for another change. List out line 100.Type a 99 over the 100 in line 100 and Edit : Tr=19:SE=10:FB=72:NB=03 (Tr=TRack(19),SE=SEctor(10),FB=First Byte Position (72), NB=NUmber of bytes (03) <EA EA EA>). Hit RETURN to lock in. Finally add a new data statement. In a clear spot, TYPE : < 1900 DATA234,234,234 > and hit RETURN. Again, save the new parm to the work disk < SAVE "TEST 2",8 > . Run the parameter on the backup again. This time you'll find it works fine. This title although not file copiable is completely void of copy protection. Note: if you are confused as to how the parameter should look after you're done, list out the Express Raider(tm) parm from the Utility disk and list it out. It may become a little clearer to you.

< < < PROGRAM: BREAKTHROUGH(tm) <> PUBLISHER: DATAEAST(tm) > > >

Procedure:

Loading the original disk produces a rattle free load, and an error scan shows no standard errors. A backup made with the C-64 Fast Copier produces a non working copy. A backup made with a nibbler produces the same non working backup. Before starting to work on this title, make a fast-copier backup and a formated work disk. Because the only file on the directory of this title is the loader, special procedures will be required. You will need a reset button of some sort (provided in Revealed Vol I).

Working with your backup:

1/ With the reset switch in place, load the backup three or four times to get the feel of when the program stalls. When you have gotten the timing down, try to reset the computer just before that stall occurs. You will hear the head swing out if you are too late. We want to reset just before it does. After reset, from the Utility disk, load the $C000 monitor < LOAD "49152",8,1 > and after the load sys it in < SYS 49152 > .

2/ If you have performed the breaks in Section E from KJ Revealed Vol I, you will remember that we are looking for a decrypter that hides the protection check. That decrypter ALWAYS begins with A0 00 A9. So we can search most of memory, flip out the BASIC Interpreter by changing memory location $0001 from a $37 to a $36 ($76 on the C-128) < M 0001 > . Now do a hunt for the key bytes in memory < H 0800 BFFF A0 00 A9 > . If you have reset out at the proper time, the following addresses will be returned: 0F13 B4ED B9E8 . Start by disassembling $0F13 < D 0F13> and inspecting the code below that address. If the code is clean, it is not what we are looking for. Inspect all the returned addresses and look for programming that has code beneath it that does not disassemble properly (usually you'll find a lot of ?????) . You'll find that $0F13 fits the bill exactly. Here you'll find the decrypter with about a sector of encrypted code beneath it.

3. Because of the no directory files problem, this break poses a slight inconvience. We will have to search the disk for the proper place to lay down the break code. This type of loader uses a Track & Sector method of loading. You'll find that each page in memory occupies its own sector on the disk. Because the break code is between $0F00 and $1100, we need to record the first 5 or 6 bytes from $0F00 and $1000 to make it easier to locate these on the disk. (Remember these will be the first bytes in the sectors they occupy. Using the Memory command, inspect and record the first few bytes in each: $0F00= 8D 5A 0D A9 81 85 02/$1000= 00 00 00 00 00 00 00. Again disassemble memory at the decrypter and use the cursor key to scroll down through memory < D 0F13 > . You must scroll down at least a full sector ($1013) and a bit more, until you see clean code again. From $1013-$1041 you'll find all zero bytes. Record this information for later referance.

Working with your original:

4/ Power off and on again to clear memory. Load the original disk until the game has started up and again hit the reset button. From the Utility Disk, again load and activate the $C000 monitor as before. Start disassembly at $0F13 < D 0F13 > . You'll find new code has replaced the previous encrypted code. The key to breaking this type of protection is to replace the encrypted code with this new code. Disassemble again at $0F13 and cursor down through memory. At $1013-$1041, you'll find the same zero bytes as we recorded earlier. This tells us that the code from here on is the same as it was in the unrun and encrypted state. Place your formatted work disk in the drive and save the new code < S "BLOCK",08.0F13,1014 > .

Working with your backup:

5/ Our task now is to transfer the code in the BLOCK file to the backup disk in the proper location. Here's the procedure. Power off and on again. Load the Disk Dr from the Utility Disk and RUN it < LOAD "DISK DOCTOR",8,1 > . Using the - command from Disk Dr, search from Track 18/Sector 0 backwards one sector at a time. You'll be looking for the Sector that contains 8D 5A 0D A9 81 85 02 as it's first seven bytes ($0F00 in Memory) and 00 00 00 00 00 00 00 as it's first seven bytes ($1000 in Memory). This search is time consuming but necessary. You will find that $0F00-$0FFF will be at Track 17/ Sector 19 and $1000-$10FF at Track 17/Sector 6. Thus the code must be placed at Track 17/Sector 19 Position $13 (19 in decimal) and continues on to Track 17/Sector 6 position $00 to end.

6/ Using Hesmon(tm), convert our BLOCK start and end addresses to decimal. $0F13 = 3859 and $1013 = 4115. Power down and remove Hesmon(tm). Now let's begin creating the parameter that will lay down the saved code in the proper location on the backup for us. Follow these instructions precisely.

a. From the work disk load the BLOCK file < LOAD "BLOCK",8,1 > .
b. Type NEW and hit RETURN.
c. From the Utility Disk load the PARM TEMPLATE < LOAD "PARM TEMPLATE",8 > .
d. List out the template and inspect. Start the data maker by typing GOTO600
e. Hit RETURN to continue. Enter Start as 3859 and END as 4115.
f. Record the number of bytes for use later (257 bytes) and hit RETURN.
g. The datamaker will now PEEK memory where our BLOCK is stored and convert the bytes to data statements in decimal.
h. When the program ends, LIST again. Edit line 5 for the desired title.
i. List out line 100 and Edit :TR=17:SE=19:FB=19:NB=237
   Tr=TRack(17),SE=SEctor(19),FB=First Byte Position (19), NB=NUmber of bytes (237) <256-19=237>. Hit RETURN to lock in.
j. Type a 101 over the 100 in line 100 and Edit : Tr=17:SE=6:FB=00:NB=20
   Tr=TRack(17),SE=SEctor(6),FB=First Byte Position (00), NB=NUmber of bytes (20) <257-237=20>.Hit RETURN to lock in.
k. Save the new parm to the work disk < SAVE "TEST",8 > .

7/ Run the parameter on the backup again. You'll find it works fine. This title although not file copiable is completely void of copy protection. Note: if you are confused as to how the parameter should look after you're done, list out the Breakthrough(tm) parm from the Utility disk and list it out. It may become a little clearer to you.

< < < INTRODUCTION: PROTECTION SCHEME TYPE F > > >

This protection scheme although tough to copy, can usually be reproduced by a few of the modern nibblers such as The Shotgun(tm). Because the protection is on one of the outter tracks (36-40), you must copy out to track 40. This scheme was developed in England and is seen on many of the Firebird(tm) releases. A few other publishers have used this scheme but those also had obvious English origins.

Characteristics of this scheme are references to GMA in the loader code or in the directory. Shortly after booting a non working copy, you can hear the head swing out and then the drive will lock up. Opening the drive door produces no flicker of the working drive light. Many times after a load failure, you will have to initialize your drive.

In short, this protection is executed at the beginning of the boot up process. It is generally accessed by a JSR to code that checks special code placed on an outter track, usually track 38. A numeric value is returned back only if the protection is in place. If a non working copy is being booted, the drive head will swing out and lock up. Some of these schemes use the numeric value brought back and some do not. We will examine three types. In all cases we will show you how to fool the code into not even doing the protection check. Also we will show you how to repair the often times corrupted directories.

Before working on these titles, please make a Fast Copy, and repair the directory.(See the general instructions below.) A disk log would also be helpful.

Using Disk Doctor and the map below, you shoud be able to repair most any directory. Let's begin with Track 18 sector 1. The first two bytes represent the link bytes. They will either point to the next track and sector or will indicate that this sector is the last one in the directory. Gennerally if more than one sector is used in the directory, you will find an rd at position Ø. This represents a link to track 18 sector 4. A @ followed by a decimal 255 represents the last sector of the directory. If when starting at track 18 sector Ø you cannot use the n key and link the directory sectors together, you will have to repair these pointer bytes. After a little practice, this task will become easy. Now for the file entries. Most changes can be made in the text mode. Program type is rairly corrupted and a @ at that position indicates a scratched file. These are normal and should remain scratched. The track and sector pointers must point to valid tracks and sectors or they are most likely dummy files meant to prevent file copying. Titles may have only upper and lower text in them. Those with text followed by other than a shifted space (decimal 16Ø) should be filled with shifted spaces. Only occasionaly will a program demand an unstandard file name. Finally, the number of sectors are not of major importance and will be normalized after file copying (when possible).

Track 18 Sector Ø represents the BAM and is often corrupted also. The main spots are position 2 which is the DOS flag byte. A byte other than an A will prevent you from writing to that disk. Change this byte if not normal using the text mode. Position 144 (decimal) represents the disk title and ID. These are in almost all cases, cosmetic and should be normalized. The title should be normal text and any unused title spaces should contain shifted spaces (decimal 160). The ID beginning at decimal position 162 can if desired, be 5 characters. These must however be normal text characters.

Maps of normal sectors have been given. Use these maps and Disk Doctor to examine our Utility Disk. When you understand the normal format, the abnormal will become easy to fix.

Track 18/Sector 0

	Title	Sh/Spaces	ID	Sh/Space	2A	Sh/Spaces
Pos:	144-159	160-161	162-163	164	165-166	167-170

Track 18/Sector 1-18

Program Type	*	Track	*	Sector	*	Title	*	# of Sectors
2	*	3	*	4	*	5-20	*	30
34	*	35	*	36	*	37-52	*	62
66	*	67	*	68	*	69-84	*	94
98	*	99	*	100	*	101-116	*	126
130	*	131	*	132	*	133-148	*	158
162	*	163	*	164	*	165-180	*	190
194	*	195	*	196	*	197-212	*	222
226	*	227	*	228	*	229-244	*	254

< < < PROGRAM: ARTIST 64$^{(tm)}$ <> PUBLISHER: WIGMORE$^{(tm)}$ > > >

Procedure:

Loading the original disk reveals the GMA symbol on the opening loader screen. A fast copy when booted, locks up the drive and sends it into an endless spin. Before starting, make a fast copy using our C-64 Fast Copy. Repair the directory according to the step one instructions and then validate the disk. Finally, a disk log may be helpful.

Working with your backup:

1/ Load Disk Doctor from the Utility Disk, and inspect Track 18/Sector 1. You should find this sector to be normal. Use the - key to go to Track 18 / Sector 0. You'll find the NAME and ID number to be corrupted. One way to repair it is as follows: Cursor to position 144 and type <t> for text mode. In this mode type ARTIST 64 followed by shifted spaces (decimal 160) to position 162. Now type AR/64 and hit RETURN. Write these changes to the backup by typing <r> followed by a <y>. Lastly while at this sector hit <n> to go to the next sector in the directory. You'll find that it goes to Track 18/Sector 4. At Track 18/Sector 4 you'll find no directory entries. The correct path is to Track 18/Sector 1 so go back to Track 18/Sector 0 and change the first two bytes from rd to ra or 18 1 in decimal. Use the @ key to make each change and be sure to rewrite your changes to the backup. Now power down and load and check the directory. The file names should be present. Validate the disk and then using the disk logger, log the file addresses.

2/ With Hesmon$^{(tm)}$ in the cartridge port, load the boot file < L"B" 08 >. At the end of the load, Disassemble code at $02A7 <D 02A7> and using the cursor down key, scroll down through memory. The code highlights are:

```
  a/ D 02C6 : JSR FF90 (control load messages)
  b/ D 02CF : JSR FFBA (set logical addresses)
  c/ D 02D8 : JSR FFBD (set file name:3 characters located at $02C1:Use
              Interpret command to see <I 02C1> the file name GM1.
  d/ D 02F1 : JSR FFD5 (load into ram)
  e/ D 02FD : JMP C000 (Jump to location $C000.)
```

3/ We now know that the next file loaded in is GM1 and that the code at $C000 is the jump link. Load the GM1 file as < L"GM1" 08 >. Start disassembly of code at $C000.

a/ Let's execute the code at $C000 and see what happens. Type <G C000>. Notice the beginning screen comes up and asks for y or n for fastloader. Type n and listen. A short load takes place and the head swings out. The drive will be locked up. Power down and up again, type X to return to basic and initialize your drive. When the drive stops, hit runstop/restore to return to the monitor.

b/ Again load the GM1 file as before and start Disassembling code at $C000 <D C000>. Cursor down through the code to $C024. Here you'll find a JSR C800. This is the actual protection check routine. Notice the next instruction is a PHA which places the numeric value returned from the protection check on the stack. This value is the key to this protection scheme.

c/ Make sure you place a write protect on the ORIGINAL Artist 64$^{(tm)}$ and place it into the drive. Using the Memory Command, change the PHA(48) at $C027 to a

BRK(00). <M C027>. We can now execute the protection code from the original and the value in the left in the A register when the code breaks will be the numeric value we're looking for. Execute the code by typing <G C000>.

d/ The opening screen will again appear and input N again and the load will continue. This time the head will swing out and a few moments later the program will break. The registers will be on the screen. Note the A register has a value of 24. This is the value we're looking for. (Those who want to inspect the drive routine that checks protection may find it starting at $C800.)

e/ The break is now quite simple. We can replace the JSR C800 instruction with the value and totally skip the protection check. By replacing it with A9 24 EA (LDA 24 NOP) we can directly load the accumulator with a 24 which then will be pushed onto the stack. Let's make our changes with Disk Doctor.

f/ Using the converter in Hesmon$^{(tm)}$, find the decimal equivalent to A9 24 EA. In a clear work space type <$ 00A9>. The decimal value 169 will be returned. The same procedure for 0024 and 00EA will return 36 and 234 respectively. Power down and remove Hesmon$^{(tm)}$. From the Utility disk, load Disk Doctor and again insert the backup into the drive. At Track 18/Sector 1, position 34, you'll find the Prg byte for the GM1 file. Place the cursor on the Track pointer at position 35 and press j to Jump to Link. You'll be taken to Track 17, Sector 1. Starting at position 0 cursor along and look for the hex bytes 20 00 C8 (JSR C800) pattern. At position 40 you'll find the first byte of that pattern. Use the @ key to change three bytes starting at position 40 to 169, 36, 234 (decimal equivalent). Hit the <r> key to rewrite the sector and then <y> for yes. Your title is now free from all protection and may even be file copied if desired.

< < < PROGRAM: COLOSSUS CHESS(tm) <> PUBLISHER: FIREBIRD(tm) > > >

Procedure:

Loading the original disk reveals the GMA symbol on the opening loader screen. A fast copy when booted, locks up the drive and sends it into an endless spin. Before starting, make a fast copy using our C-64 Fast Copy. Repair the directory according to the step one instructions. Be sure to validate the disk and do a log the disk as a part of your preparation.

Working with your backup:

1/ Load Disk Doctor from the Utility Disk, and inspect Track 18/Sector 1. You should find this sector to be normal. Use the - key to go to Track 18 / Sector 0. You'll find the NAME and ID number to be corrupted. One way to repair it is as follows: Cursor to position 144 and type <t> for text mode. In this mode type COLOSSUS followed by shifted spaces (decimal 160) to position 162. Now type CHESS and hit RETURN. Write these changes to the backup by typing <r> followed by a <y>. Lastly while at this sector hit <n> to go to the next sector in the directory. You'll find that it goes to Track 18/Sector 1. Continue hitting n to go to each linked sector in the directory. You'll find every sector to be normal with the last directory sector at Track 18/ Sector 5. Now power down and load and check the directory. The file names should be present. Validate the disk and then using the disk logger, log the file addresses.

2/ With Hesmon(tm) in the cartridge port, load the boot file < L"Firebird" 08 >. At the end of the load, Disassemble code at $02A7 <D 02A7> and using the cursor down key, scroll down through memory. The code highlights are:

a/ D 02C6 : JSR FF90 (control load messages)
b/ D 02CF : JSR FFBA (set logical addresses)
c/ D 02D8 : JSR FFBD (set file name:3 characters located at $02C1:Use
Interpret command to see <I 02C1> the file name GM1.
d/ D 02F1 : JSR FFD5 (load into ram)
e/ D 02FD : JMP 0334 (Jump to location $0334.)

3/ We now know that the next file loaded in is GM1 and that the code at $0334 is the jump link. Load the GM1 file as < L"GM1" 08 >. Notice that the code fills the screen. This is because it is loaded into screen memory. Cursor down and start disassembly of code at $0334.

a/ Disassemble code at $0334 <D 0334>. You'll find a jump to $034B. Cursor down and inspect the code from $034B-$036B. This code represents the key to the protection. This particular code can be found in many similar titles and the break for all is about the same. This code sets up a load of the actual protection check code within the GMA3 file. (Those of you interested in the drive code for the protection should load and inspect GMA3.) A JSR to $C800 within this code checks protection, and if the check is successful, a value of $97 is place at computer location $0002. Upon return from the JRS C800, the value in location $0002 is loaded into the accumulator and EORed with a value of $97. Lastly the code Branches if Equal(to 0) to $036C. Remember, if protection WAS satisfied, a value of $97 was placed at $0002. The EOR Truth Table in the back of the book tells us that $97 EORed with $97 is in fact zero. If the branch does take place, it will cause a jump around the instruction at $0369 which is a

JMP ($FFFC). This instruction is actually a Jump to a Kernal routine that does a system reset, which in turn will crash the load process.

b/ The break is now quite simple. We can jump around the whole protection check. All that is necessary is to replace the JSR C800 with a JUMP around the reset code to $036C. We will replace the 20 00 C8 with 4C 6C 03 (JMP 036C). Remember, we don't want to allow any protection check because if the protection is not in place, the drive hangs up and goes into an endless spin. Let's make our changes with Disk Doctor.

c/ Using the converter in Hesmon$^{(tm)}$, find the decimal equivalent to 4C 6C 03. In a clear work space type <$ 004C>. The decimal value 75 will be returned. The same procedure for 006C and 0003 will return 108 and 03 respectively. Power down and remove Hesmon$^{(tm)}$. From the Utility disk, load Disk Doctor and again insert the backup into the drive. At Track 18/Sector 1, position 34, you'll find the Prg byte for the GM1 file. Place the cursor on the Track pointer at position 35 and press <j> to Jump to Link. You'll be taken to Track 17, Sector 1. Starting at position 0 cursor along and look for the hex bytes 20 00 C8 (JSR C800) pattern. At position 48 you'll find the first byte of that pattern. Use the @ key to change three bytes starting at position 48 to 76, 108, 03 (decimal equivalent). Hit the <r> key to rewrite the sector and then <y> for yes. Your title is now free from all protection and may even be file copied if desired.

< < < PROGRAM: COMPUTER SCRABBLE$^{(tm)}$<>PUBLISHER: LEISURE GENIUS$^{(tm)}$ > > >

Procedure:

Loading the directory of the original disk reveals the GMA symbol. A fast copy when booted, locks up the drive and sends it into an endless spin. Before starting, make a fast copy using our C-64 Fast Copy. Repair the directory according to the step one instructions. Be sure to validate the disk and do a log the disk as a part of your preparation.

Working with your backup:

1/ Load Disk Doctor from the Utility Disk, and inspect Track 18/Sector 1. You should find this sector to be normal. Use the - key to go to Track 18 / Sector 0. You'll find the NAME and ID number to be corrupted. One way to repair it is as follows: Cursor to position 144 and type <t> for text mode. In this mode type SCRABBLE followed by shifted spaces (decimal 160) to position 162. Now type LG/SC and hit RETURN. Write these changes to the backup by typing <r> followed by a <y>. Lastly while at this sector hit <n> to go to the next sector in the directory. You'll find that it goes to Track 18/Sector 1. Continue hitting n to go to each linked sector in the directory. You'll find every sector to be normal with the last directory sector at Track 18/ Sector 4. Now power down and load and check the directory. The file names should be present. Validate the disk and then using the disk logger, log the file addresses.

2/ With Hesmon$^{(tm)}$ in the cartridge port, load the boot file < L"B" 08 >. At the end of the load, Dissasemble code at $02A7 <D 02A7> and using the cursor down key, scroll down through memory. The code highlights are:

a/ D 02C6 : JSR FF90 (control load messages)
b/ D 02CF : JSR FFBA (set logical addresses)
c/ D 02D8 : JSR FFBD (set file name:3 characters located at $02C1:Use
Interpret command to see <I 02C1> the file name GM1.
d/ D 02F1 : JSR FFD5 (load into ram)
e/ D 02FD : JMP 3800 (Jump to location $3800.)

3/ We now know that the next file loaded in is GM1 and that the code at $3800 is the jump link. Load the GM1 file as < L"GM1" 08 >. Start disassembly of code at $3800.

a/ Let's execute the code at $3800 and see what happens. Type <G 3800>. Notice the beginning screen comes up and asks for y or n for fastloader. Type n and listen. A short load takes place and the head swings out. The drive will be locked up. Power down and up again, type X to return to basic and initialize your drive. When the drive stops, hit runstop/restore to return to the monitor.

b/ Again load the GM1 file as before and start Disassembling code at $3800 <D 3800>. Cursor down through the code to $384A. Here you'll find a JSR C800. This is the actual protection cneck routine. Notice the next instruction is a PHA which places the numeric value returned from the protection check on the stack. This value is the key to this protection scneme.

c/ Make sure you place a write protect on the ORIGINAL Artist 64$^{(tm)}$ and place it into the drive. Using the Memory Command, change the PHA(48) at $384D to a BRK(00). <M 384D>. We can now execute the protection code from the original and the value in the left in the A register when the code breaks will be the numeric value we're looking for. Execute the code by typing <G 3800>.

d/ The opening screen will again appear and input N again and the load will continue. This time the head will swing out and a few moments later the program will break. The registers will be on the screen. Note the A register has a value of 58. This is the value we're looking for. (Those who want to inspect the drive routine that checks protection may find it starting at $C800.)

e/ The break is now quite simple. We can replace the JSR C800 instruction with the value and totally skip the protection check. By replacing it with A9 58 EA (LDA 58 NOP) we can directly load the accumulator with a 58 which then will be pushed onto the stack. Let's make our changes with Disk Doctor.

f/ Using the converter in Hesmon(tm), find the decimal equivalent to A9 58 EA. In a clear work space type <$ 00A9>. The decimal value 169 will be returned. The same procedure for 0058 and 00EA will return 88 and 234 respectively. Power down and remove Hesmon(tm). From the Utility Disk, load Disk Doctor and again insert the backup into the drive. At Track 18/Sector 1, position 34, you'll find the Prg byte for the GM1 file. Place the cursor on the Track pointer at position 35 and press <j> to Jump to Link. You'll be taken to Track 17, Sector 1. Starting at position 0 cursor along and look for the hex bytes 20 00 C8 (JSR C800) pattern. At position 78 you'll find the first byte of that pattern. Use the @ key to change three bytes starting at position 78 to 169, 88, 234 (decimal equivalent). Hit the <r> key to rewrite the sector and then <y> for yes. Your title is now free from all protection and may even be file copied if desired.

< < < PROGRAM: FALKLANDS 82$^{(tm)}$ <> PUBLISHER: FIREBIRD$^{(tm)}$ > > >

Procedure:

Loading the original disk reveals the GMA symbol on the opening loader screen. A fast copy when booted, locks up the drive and sends it into an endless spin. Before starting, make a fast copy using our C-64 Fast Copy. Repair the directory according to the step one instructions. Be sure to validate the disk and do a log the disk as a part of your preparation.

Working with your backup:

1/ Load Disk Doctor from the Utility Disk, and inspect Track 18/Sector 1. You should find this sector to be normal. Use the - key to go to Track 18 / Sector 0. You'll find the NAME and ID number to be corrupted. One way to repair it is as follows: Cursor to position 144 and type <t> for text mode. In this mode type FALKLANDS followed by shifted spaces (decimal 160) to position 162. Now type FL/82 and hit RETURN. Write these changes to the backup by typing <r> followed by a <y>. Now take a look at position 2. Anything other than a capital A in that spot will prevent you from writing to the disk. You must cursor up to position 2 and hit <t> for text mode then type A to that position. Hit <r> for rewrite and <y> for yes. Lastly while at this sector hit <n> to go to the next sector in the directory. You'll find that it goes to Track 18/Sector 1. Continue hitting n to go to each linked sector in the directory. You'll find every sector to be normal with the last directory sector at Track 18/ Sector 4. Now power down and load and check the directory. The file names should be present. Validate the disk and then using the disk logger, log the file addresses.

2/ With Hesmon$^{(tm)}$ in the cartridge port, load the boot file < L"Firebird" 08 >. At the end of the load, Disassemble code at $02A7 <D 02A7> and using the cursor down key, scroll down through memory. The code highlights are:

```
  a/ D 02C6 : JSR FF90 (control load messages)
  b/ D 02CF : JSR FFBA (set logical addresses)
  c/ D 02D8 : JSR FFBD (set file name:3 characters located at $02C1:Use
              Interpret command to see <I 02C1> the file name GM1.
  d/ D 02F1 : JSR FFD5 (load into ram)
  e/ D 02FD : JMP C000 (Jump to location $C000.)
```

3/ We now know that the next file loaded in is GM1 and that the code at $C000 is the jump link. Load the GM1 file as < L"GM1" 08 >. Cursor down and start disassembly of code at $C000.

a/ Disassemble code at $C000 <D C000>. You'll find a Jump to $C00F. Cursor down and inspect the code from $C00F-$C029. This code represents the key to the protection. This particular code can be found in many similar titles and the break for all is about the same. This code sets up a load of the actual protection check code within the GMA3 file. (Those of you interested in the drive code for the protection should load and inspect GMA3.) A JSR to $C800 within this code checks protection, and if the check is successful, a value of $97 is place at computer location $0002. Upon return from the JRS C800, the value in location $0002 is loaded into the accumulator and EORed with a value of $97. Lastly the code Branches if Equal(to 0) to $C02A. Remember, if protection WAS satisfied, a value of $97 was placed at $0002. The EOR Truth Table in the back of the book tells us that $97 EORed with $97 is in fact zero. If the branch does take place, it will cause a jump around the instruction at $C027 which is a JMP ($FFFC). This instruction is actually a Jump to a Kernal routine that does a system reset, which in turn will crash the load process.

b/ The break is now quite simple. We can jump around the whole protection check. All that is necessary is to replace the JSR C800 with a JUMP around the reset code to $C02A. We will replace the 20 00 C8 with 4C 2A C0 (JMP C02A). Remember, we don't want to allow any protection check because if the protection is not in place, the drive hangs up and goes into an endless spin. Let's make our changes with Disk Doctor.

c/ Using the converter in Hesmon(tm), find the decimal equivalent to 4C 2A C0. In a clear work space type <$ 004C>. The decimal value 76 will be returned. The same procedure for 002A and 00C0 will return 42 and 192 respectively. Power down and remove Hesmon(tm). From the Utility Disk, load Disk Doctor and again insert the backup into the drive. At Track 18/Sector 1, position 34, you'll find the Prg byte for the GM1 file. Place the cursor on the Track pointer at position 35 and press j to Jump to Link. You'll be taken to Track 17, Sector 1. Starting at position 0 cursor along and look for the hex bytes 20 00 C8 (JSR C800) pattern. At position 34 you'll find the first byte of that pattern. Use the @ key to change three bytes starting at position 34 to 76, 42, 192 (decimal equivalent). Hit the r key to rewrite the sector and then y for yes. Your title is now free from all protection and may even be file copied if desired.

< < < INTRODUCTION: PROTECTION SCHEME TYPE G > > >

Most computer software houses utilitze some form of "copy protection" that prevents the average consumer from making backup copies of the program(s) that the company distributes. Even the most basic Commodore$^{(tm)}$ user is aware that protection is included on most of the commercial programs he buys. Using a simple data-copier to archive the original usually fails to make a working copy. One company on the other hand, uses a different approach for their latest series of sports games. Instead of encoding the protection upon the diskette where the game is stored, included with the sale of each of their programs is a device called a "dongle". The dongle is simply a small plastic device that plugs into the cassette port of your Commodore 64/128$^{(tm)}$. The dongle includes a small resistor that makes it look complicated, but it is actually a very simple device. The resistor merely ties a positive 6 volt lead to an input port that the Commodore$^{(tm)}$ uses for cassette load/save interfacing. The fact is, the resistor on the dongle could be replaced with a simple piece of wire. The resistor serves merely either to avoid "shorting" out your Commodore$^{(tm)}$ (which is doubtful), or, as most of us tend to see it, as a deceiving device. Through software, the programmer checks a certain memory location to see if that particular bit has a 0 value (dongle in place), or a 1 value (dongle not plugged in.) If the bit value retrieved is a "1", the program refuses to operate.

The following tutorials will deal with deprotecting the software checks in the program code. Looking through machine-language code for a protection-check is quite a time-consuming task since there are probably a million ways to check if a bit value at a certain memory location is either on or off. In the following pages, we will try to give you some of the more popular methods.

The bit that the dongle triggers is located at memory location $0001. Using a machine-language monitor, we can verify that bit 4 is always on without the dongle plugged in.

$0001:

Bit	7	6	5	4	3	2	1	0
	X	X	X	1	X	X	X	X

Bit 4 will become "0" when the dongle is plugged in. A short machine-code program assembled in the cassette buffer ($0334) can check the 4th bit:

```
A 0334  LDA #$10
A 0336  BIT $01
A 0338  BEQ $033A
A 0339  BRK
A 033A  BRK
```

Type G 0334 with the dongle in or out.

The BIT instruction "AND's" memory location $01 with the value in the accumulator (#$10 = check bit 4). If the dongle is plugged in, both bits will match up (both 1's), and the branch instruction will be bypassed and the program will break into the monitor at $0339.

Running the program again with the dongle plugged in will AND a 1 bit with the dongle 0 bit, causing the branch to be executed. The program will break into the monitor at $033A. This is just one method in which ACCESS checks their protection. We can "break" their protection checks by replacing LDA #$10 with LDA #$00. This way, the BIT insruction will always result in setting the zero flag, which emulates the dongle!

Here are some other code forms for checking the dongle:

```
LDA #$10
BIT $00     (memory location zero, bit 4 holds an image of $0001)
BEQ dongle in
```

Solution: replace LDA #$10 with LDA #$00.

```
LDA $01
AND #$10
BEQ dongle in
```

Solution: replace AND #$10 with AND #$00.

```
LDA #$40
LSR
LSR
TAX
AND $FFF1,X
BEQ dongle in
```

Solution: replace LDA #$40 with LDA #$00.

```
LDA $0001
ASL
TAX
ASL
ASL
ASL
BCS dongle out
```

Solution: replace BCS with two "NOP"'s.

There are many other ways to check memory location $0001 for the dongle bit. In the following pages you will find instructions on how to disable the checks in four programs. These should give you the insight necessary to continue on your own.

< < < PROGRAM: LEADERBOARD(tm) <> PUBLISHER: ACCESS(tm) > > >

Procedure:

Use the C-64 Fast Copier utility to make an exact data-copy of the original. This backup will run like the original ONLY if the dongle is in place. The following procedure will eliminate all dongle-checks:

Working with your backup:

1/ Turn on your computer and from the Utility Disk, load the Disk Logger by typing < LOAD "DISK LOGGER",8 > . Then type RUN. Insert your backup copy of Leaderboard(tm) in the drive and log it. The two files on the disk that contain code that check for the dongle are called "L" and "H". Take note of the addresses in memory where these programs reside:

```
   "L"   $081D - $3E32
   "H"   $9280 - $AB9A
```

This information is important since we need to load a machine-language monitor into memory where these programs aren't! We can choose from one of three monitors ($2000 = 8192, $8000 = 32768, or $C000 = 49152). The monitor at $C000 does not conflict with Leaderboard(tm) memory, so let's use it.

2/ Turn on the computer again and load the $C000 monitor from your utility disk < LOAD "49152",8,1 > followed by < SYS 49152 > to execute it.

3/ To start with a clean slate, let's clear out all memory below the monitor by typing < F 0800 BFFF EA > .

4/ From the monitor, we must load the two Leaderboard(tm) files. Insert your backup copy in the drive and load both files: < L "H",08 > and < L "L",08 > .

5/ Since the "H" file resides in the RAM underneath the BASIC ROMS ($A000-$BFFF), we have to use the bank select bits to bank out the ROM and bank in the RAM so we can view the "H" file code. Using the memory command, change location $0001 to 36 (76 on the 128) < M 0001 >.

6/ Now, we will began searching for the certain "dongle-check" byte sequences. We can use the monitor "H" command to hunt through memory for these patterns. Type < H 0800 BFFF A9 10 24 01 > . After a brief wait, the monitor should return addresses: 0AA2 112F A03C.

7/ Disassemble each of these addresses using the < D > command. Use the cursor-down key to scroll through the next couple of addresses. At the top after each assembly, change the LDA #$10 command to: LDA #$00 (see intro). i.e. - < D 0AA2 >, < A 0AA2 LDA #$00 >... do the same for the other two addresses. The rest of the byte changes are performed in this manner, so they won't be in detail.

8/ Type < H 0800 BFFF A9 40 4A 4A AA > Monitor finds: 1245 9D20.

9/ Disassemble both addresses, and change the LDA #$40 command to LDA #$00 (see intro).

10/ Type < H 0800 BFFF AD 01 00 > .Monitor finds: 9AE0.

11/ Disassemble $9AE0 and cursor down 10 or 11 times. Find the BCS instruction and replace it with two NOPs (see intro). < A 9AE8 NOP > < A 9AE9 NOP > .

12/ Type < H 0800 BFFF 58 FF > . Monitor finds: 14D1 A6F4.

13/ First, disassemble a few bytes before $14D1. say at $14C0. You will discover a routine that looks something like the following:

```
LDX #$09
LDA $14D8,X
EOR #$FF
STA $FF58,X
```

Notice that this routine decrypts a sequence of bytes beginning at $14D8 by EOR'ing it with the value of #$FF and stores it in hi-memory hidden beneath the Kernal ROMs. The routine itself breaks into the IRQ routine and checks the dongle bit every time the IRQ routine pointed to by vector $0314-$0315 is executed. To see the decrypted code, you will have to point the routine to a location in RAM that is easily visable, say $0801 (FF58 = 0801). If you do, be sure to start the break procedure over, for you will have corrupted our work up to now.

14/ To "trick" the routine into thinking that the dongle is always in, type < M 14D8 > . The monitor should return a sequence of 8 bytes.

15/ Edit the 4th byte over (should be $EF) and change it to $FF.

16/ Next, disassemble memory a few bytes before $A6F4 by typing < D A6F0 > . Use cursor/down to display the next 14 or 15 bytes. The monitor should show you something like:

```
LDX #$09
CLC
ADC $FF58,X
DEX
```

17/ This group of instructions is simply a checksum check of the IRQ dongle-check routine we just finished working with. In other words, they are "double-checking" their protection code. Find the instruction that compares the checksum value in the accumulator with a set value. Notice the 'BEQ' immediately afterwards that bypasses protection failure. Simply change 'CMP #$5A' with 'LDA #$00'. We have just set the zero flag permanently, and the routine is tricked".

18/ Now that we have finished removing all the dongle-check routines, we need to re-save the two files to your backup disk. Type:

```
< S"@0:L",08,081D,3E33 >
< S"@0:H",08,9280,AB9B >
```

19/ You now have a dongle-free backup of Leaderboard$^{(tm)}$. It may be archived using any simple datacopier. Note: The parameter LEADERB. PARM 1 represents this particular break method. LEADERB. PARM 2 is a variation of this break and can be run on a backup and examined with the monitor.

< < < PROGRAM: EXECUTIVE LEADERBOARD #1(tm) <> PUBLISHER: ACCESS(tm) > > >

Procedure:

Use the C-64 Fast Copier utility to make an exact data-copy of the original. This backup will run like the original ONLY if the dongle is in place. The following procedure will eliminate all dongle-checks:

Working with your backup:

1/ Turn on your computer and from the Utility Disk, load the Disk Logger by typing < LOAD "DISK LOGGER",8 > . Then type RUN. Insert your backup copy of Executive Leaderboard #1(tm) in the drive and log it. The two files on the disk that contain code that check for the dongle are called "L" and "H". Take note of the addresses in memory where these programs reside:

```
"L"   $081D - $3FAF
"H"   $9280 - $BAEC
```

This information is important since we need to load a machine-language monitor into memory where these programs aren't! We can choose from one of three monitors ($2000 = 8192, $8000 = 32768, or $C000 = 49152). The monitor at $C000 does not conflict with Exec Leaderboard #1(tm) memory, so we will use it.

2/ Turn on the computer again and load the $C000 monitor from your utility disk < LOAD "49152",8,1 > followed by < SYS 49152 > to execute it.

3/ To start with a clean-slate, let's clear out all memory below the monitor by typing < F 0800 BFFF EA > .

4/ From the monitor, we must load the two Exec Leaderboard #1tm) files. Insert your backup copy in the drive and load both files: < L "H",08 > and < L "L",08 > .

5/ Since the "H" files resides in the RAM underneath the BASIC ROMS ($A000-BFFF), we have to use the bank select bits to bank out the ROM and bank in the RAM so we can view the "H" file code. Using the memory command, change memory location $0001 to 36 (76 on the 128) < M 0001 > .

6/ Now, we will began searching for the certain "dongle-check" byte sequences. We can use the monitor "H" command to hunt through memory for these patterns. Type < H 0800 BFFF A9 10 24 01 > . After a brief wait, the monitor should return addresses: 0A9C 1114 9FA2.

7/ Disassemble each of these addresses using the "D" command. Use the cursor-down key to scroll through the next couple of addresses. At the top after each assembly, change the LDA #$10 command to: LDA #$00 . i.e. - < D 0A9C > , < A 0A9C LDA #$00 > ... do the same for the other two addresses. The rest of the byte changes are performed in this manner, so they won't be in detail!

8/ Type < H 0800 BFFF A9 40 4A 4A AA > . Monitor finds: 1237 9D3E.

9/ Disassemble both addresses, and change the LDA #$40 command to LDA #$00 .

10/ Type < H 0800 BFFF A9 10 24 00 > . Monitor finds: 93EF.

11/ Disassemble and change LDA #$10 to LDA #$00.

12/ Type < H 0800 BFFF AD 0E C2 0A AA > . Monitor finds: 9AFE.

13/ Disassemble $9AFE and scroll down 6 or 7 times. Find the BCS instruction and replace it with two NOPs. < A 9B06 NOP > , < A 9B07 NOP > .

14/ Type < H 0800 BFFF 58 FF > . Monitor finds: 14AC A5E7.

15/ First, disassemble a few bytes before $14AC, say at $14A3. You will discover a routine that looks something like the following:

```
LDX #$09
LDA $14B3,X
EOR #$FF
STA $FF58,X
```

Notice that this routine decrypts a sequence of bytes beginning at $14B3 by EOR'ing it with the value of #$FF and stores it in hi-memory hidden beneath the Kernal ROMs. The routine itself breaks into the IRQ routine and checks the dongle bit every time the IRQ routine pointed to by vector $0314-0315 is executed. To see the decrypted code, you will have to point the routine to a location in RAM that is easily visable, say $0801 (FF58 = 0801). If you do, be sure to start the break procedure over, for you will have corrupted our work up till now.

16/ To "trick" the routine into thinking that the dongle is always in, type < M 14B3 > . The monitor should return a sequence of 8 bytes.

17/ Edit the 4th byte over (should be $EF) and change it to $FF.

18/ Next, disassemble memory a few bytes before $A5E7 by typing < D A5E1 > . Use cursor-down to display the next 14 or 15 bytes. The monitor should show you something like:

```
LDX #$09
CLC
ADC $FF58,X
DEX
```

19/ This group of instructions is simply a checksum check of the IRQ dongle-check routine we just finished working with. In other words, they are "double-checking" their protection code. Find the instruction that compares the checksum value in the accumulator with a set value. Notice the BEQ immediately afterwards that bypasses protection failure. Simply change CMP #$5A with LDA #$00 . We have just set the zero flag permanently, and the routine is tricked.

20/ Now that we have finished removing all the dongle-check routines, we need to re-save the two files to your backup disk. Type:

```
< S"@0:L",08,081D,3FB0 >
< S"@0:H",08,9280,BAED >
```

21/ The exact same procedure described above must be repeated for two files "L5" and "H5", which are identical other than name to "L" and "H". So repeat steps 3-20 but use "L5" and "H5" as filenames instead!

22/ After this is done, you will have a dongle-free backup of Executive Leaderboard #1(tm). It may be archived using any simple datacopier. Note: The parameter for LB Exec #1 on the utility disk represents a variation of this break and can be run on a backup and examined with the monitor. You'll find all changes in about the same memory locations.

< < < PROGRAM: LEADERBOARD TOURNAMENT DISK #1(tm) <> PUBLISHER: ACCESS(tm) > > >

Procedure:

Use the C-64 Fast Copier to make an exact data-copy of the original. This backup will run like the original ONLY if the dongle is in place. The following procedure will eliminate all dongle-checks:

Working with your backup:

1/ Turn on your computer and from the Utility Disk, load the Disk Logger utility by typing < LOAD "DISK LOGGER",8 > . Then type RUN. Insert your backup copy of Leaderboard Tournament(tm) in the drive and log it. The file on the disk that contains the code that checks for the dongle is called "B". Take note of the addresses in memory where this program resides:

```
  "B"   $9280 - $BF53
```

This information is important since we need to load a machine-language monitor into memory where this program isn't! We can choose from one of three monitors ($2000 = 8192, $8000 = 32768, or $C000 = 49152). The monitor at $C000 does not conflict with Leaderboard(tm) memory, so we will use it.

2/ Turn on the computer again and load the $C000 monitor from your utility disk < LOAD "49152",8,1 > followed by < SYS 49152 > to execute it.

3/ To start with a clean-slate, let's clear out all memory below the monitor by typing < F 0800 BFFF EA > .

4/ From the monitor, we must load the Leaderboard(tm) file. Insert your backup copy in the drive and load the file: < L"B",08 > .

5/ Since the "B" file resides in the RAM underneath the BASIC ROMS ($A000-BFFF), we have to use the bank select bits to bank out the ROM and bank in the RAM so we can view the "B" file code. Using the memory command, change memory location $0001 to 36 (76 on the 128) < M 0001 > .

6/ Now, we will began searching for the certain "dongle-check" byte sequences. We can use the monitor "H" command to hunt through memory for these patterns. Type < H 9000 BFFF A9 10 24 01 > . After a brief wait, the monitor should return address: A03C

7/ Disassemble this address using the "D" command. Use the cursor-down key to scroll through the next couple of addresses. At the top, change the LDA #$10 command to: LDA #$00 . i.e. - < D A03C > , < A A03C LDA #$00 > . The rest of the byte changes are performed in this manner, so they won't be in detail!

8/ Type < H 9000 BFFF A9 40 4A 4A AA > . Monitor finds: 9D20.

9/ Disassemble and change the LDA #$40 command to LDA #$00 .

10/ Type < H 9000 BFFF A9 10 24 00 > . Monitor finds: 93EF.

11/ Disassemble and change LDA #$10 to LDA #$00 .

12/ Type < H 9000 BFFF AD 01 00 0A AA > . Monitor finds: 9AE0.

13/ Disassemble $9AE0 and scroll down 6 or 7 times. Find the BCS instruction and replace it with two NOP's. < A 9AE8 NOP >,< A 9AE9 NOP > .

14/ Type < H 9000 BFFF 58 FF > . Monitor finds: A6F4.

15/ Diassemble memory a few bytes before $A6F4 by typing < D A6 0 > . Use cursor-down to display the next 14 or 15 bytes. The monitor should show you something like:

```
LDX #$09
CLC
ADC $FF58,X
DEX
```

16/ This group of instructions is simply a checksum check of the IRQ dongle-check routine we worked with in the Leaderboard(tm) portion of this manual. In other words, they are "double-checking" their protection code. Find the instruction that compares the checksum value in the accumulator with a set value. Notice the BEQ immediately afterwards that bypasses protection failure. Simply change CMP #$5A with LDA #$00 . We have just set the zero flag permanently, and the routine is tricked.

17/ Now that we have finished removing all the dongle-check routines, we need to re-save the file to your backup disk. Type:
< S"@0:B",08,9280,BF54 > .

18/ Now that you have removed all the dongle-check routines, you have a dongle-less working copy of Leaderboard Tournament Disk #1(tm). It may be backed-up with any datacopier. Note: the parameter for LB Tourn #1 on the Utility Disk represents a variation of this break and can be run on a backup and examined with the monitor. You'll find all changes in about the same memory locations.

< < < PROGRAM: TENTH FRAME$^{(tm)}$ <> PUBLISHER: ACCESS$^{(tm)}$ > > >

Procedure:

Use the C-64 Fast Copier to make an exact data-copy of the original. This backup will run like the original ONLY if the dongle is in place. The following procedure will eliminate all dongle-checks.

Working with your backup:

1/ Turn on your computer and from the Utility Disk, load the Disk Logger by typing < LOAD "DISK LOGGER",8 > . Then type < RUN > . Insert your backup copy of Tenth Frame$^{(tm)}$ in the drive and log it. The two files on the disk that contain code that check for the dongle are called "L" and "S". Take note of the addresses in memory where these programs reside:

```
"L"   $081D - $3FFE
"S"   $6E00 - $9FFE
```

This information is important since we need to load a machine-language monitor into memory where these programs aren't! We can choose from one of three monitors ($2000 = 8192, $8000 = 32768, or $C000 = 49152). The monitor at $C000 does not conflict with Tenth Frame$^{(tm)}$ memory, so we will use it.

2/ Turn on the computer again and load the $8000 monitor from your utility disk < LOAD "49152",8,1 > followed by < SYS49152 > to execute it.

3/ To start with a clean slate, let's clear out all memory below the monitor by typing < F 0800 BFFF EA > .

4/ From the monitor, we must load the two Tenth Frame$^{(tm)}$ files. Insert your backup copy in the drive and load both files: < L"H",08 > and < L"S",08 > .

5/ Now, we will began searching for the certain "dongle-check" byte sequences. We can use the monitor "H" command to hunt through memory for these patterns. Type < H 0800 9FFF A9 10 24 01 > . After a brief wait, the monitor should return addresses: 0F66 17D0.

6/ Disassemble each of these addresses using the < D > command. Use the <cursor-down> key to scroll through the next couple of addresses. At the top after each assembly, change the LDA #$10 command to: LDA #$00 . i.e. - < D 0F66 >,< A 0F66 LDA #$00 >.. do the same for the other address. The rest of the byte changes are performed in this manner, so they won't be in detail!

7/ Type < H 0800 9FFF A9 40 4A 4A AA > . Monitor finds: 0FF9 16E6

8/ Disassemble both addresses, and change the LDA #$40 command to LDA #$00 .

9/ Type < H 0800 9FFF A9 10 25 01 > . Monitor finds 162B 1E3D.

10/ Disassemble both addresses and change LDA #$10 to LDA #$00 .

11/ Type < H 0800 9FFF A5 01 29 10 > . Monitor finds 0EEC 11CA 1C5A 2C85 3141.

12/ Disassemble each address and change AND #$10 to AND #$00 .

13/ Type < H 0800 9FFF A9 10 24 00 > . Monitor finds 1227.

14/ Disassemble and change LDA #$10 to LDA #$00 .

15/ Type < H 0800 9FFF A9 08 0A EA 31 2B > . Monitor finds 2BB6.

16/ Disassemble and change AND ($2B),Y to AND #$00 .

17/ Type < H 0800 9FFF A9 D0 49 FF D1 2B > . Monitor finds 2C37.

18/ Disassemble the next 9 or 10 bytes. Find the BEQ instruction and replace the next instruction immediately after it with an RTS :< A 2C3F RTS > . The BEQ instruction is executed if the dongle is in, and it hits an RTS too, so putting another RTS after the BEQ guarantees that the program will not crash with the dongle out.

19/ Type < H 0800 9FFF 18 A9 00 7D 00 C0 > . Monitor finds 6EC2.

20/ Disassemble $6EC2 and scroll down 15 or 16 instructions. Find the BEQ instruction and replace the next instruction after it with an RTS again: < A 6ED9 RTS > .(We just fixed a dongle-related checksumming problem.)

21/ Now that all the dongle-check routines in these files have been removed, we need to re-save the two files to our backup:
< S"@0:L",08,081D,3FFF >
< S"@0:S",08,6E00,9FFF >

22/ There are two other dongle-routines that need to be changed on the Tenth Frame$^{(tm)}$ disk. The only problem is that they reside in a file called "P", which loads underneath the KERNAL ROM. There is no way to use our monitors to view, change and re-save this file. Instead, load the Disk Doctor utility by typing < LOAD "DISK D*",8 > and then < RUN > .

23/ Insert your backup copy of Tenth Frame$^{(tm)}$ and press <RETURN>. Use the < B > command to read Track 20, Sector 1. Use the cursor keys to move to position 63. Using the < @ > command, change the byte value (48) to 32. Re-write the sector with the < r > command. Then use < + > key to read Track 20, Sector 2. Move cursor to position 150. Change byte value (15) to 7. Re-write the sector with the < r > command.

24/ Let's investigate why we made the changes in the "P" file. From the $C000 (49152) monitor, load the "P" file from your backup copy by typing <> L"P",08.

25/ Since the "P" file resides in memory from $E000-$FEBF, it is now residing in the RAM that is "hidden" beneath the KERNAL ROM's ($E000-$FFFF). Our monitor won't let us view the RAM, so we need to write a short ML program to transfer $E000-$FFFF down to lower memory from $4000-$5FFF so we can look at the "P" code.

Type in the following routine starting at $0334:

```
A 0334 SEI
A 0335 LDA #$35
A 0337 STA $01
A 0339 LDX #$00
A 033B LDA $E000,X
A 033E STA $4000,X
A 0341 INX
A 0342 BNE $033B
A 0344 INC $0340
A 0347 INC $033D
A 034A BNE $033B
A 034C LDA #$37
A 034E STA $01
A 0350 BRK
```

25/ Type <> G 0334 to execute the routine.

26/ Type <> D 550E and cursor-down a few bytes. You should see a dongle-check routine that looks like:

```
550E   LDA $01
       AND #$30
       ORA #$8C
       STA $3A4E
```

The byte we changed using the DISK DOCTOR on track 20, sector 1 changed the "AND #$30" instruction to "AND #$20". This permanently masks out the dongle-bit to a "0" value, so the computer "thinks" that the dongle is actually in place.

27/ Type <> D 575D and cursor-down a few bytes. You should see:

```
575D   LDA $01
       LSR
       STA $A01F
       AND #$0F
       STA $A027
```

The byte we changed on track 20, sector 2 changed the "AND #$0F" instruction to "AND #$07." This also masks out the dongle bit from location $01 to appear to be on (0 bit).

28/ After all changes have been made, your Tenth Frame(tm) disk is completely broken and the dongle is no longer neccessary.

< < < RAPIDLOK(tm) PROTECTION REVEALED > > >

Most Commodore(tm) users are aware of the standard format that the 1541/1571 disk drives read. We can load and save programs, directory the disk, and perform a variety of other commands. The program code that knows how to execute all these functions is stored within the ROM's of the disk drive. Most Disk-drive Operating Systems are called "DOS". RAPIDLOK(tm)(from now on called RL) is a recent protection scheme that has appeared on the disks of some recent big-name producers (Accolade(tm), Avalon Hill(tm), Microprose(tm)...),and uses its own "DOS" system to load files. RL disks will usually have only track 18 standardly formatted, the rest of the tracks being formatted in the RL manner. The RL DOS resides in an encoded format on track 18, sectors 18,15,12,9,6, and 3. Each time a file is loaded through RL, a short machine-language auto-boot file loads the RL DOS from track 18 and stores it in the disk-drive memory from $0300-07FF. Currently, we know of 6 different versions of RL DOS. Each relies on the same basic track formatting, but in addition to loading RL files, they do a complicated check on certain sync lengths, header lengths, and track to track alignment.

RAPIDLOK FORMAT

Like Commodore(tm) DOS, RL formats its tracks by first writing a header block, and then a $0255 byte long data block. The method through which RL converts this data into REAL bytes is much too confusing to explain in this overview. The following is how RL would format one track:

1/ The Reference Header:

The first header on a RL track is the track reference header. It is actually a normal Commodore(tm) DOS header for that track, sector 0 in GCR format. It is written with a SYNC LENGTH of $0029 bytes. If RL DOS detects a reference header without the correct sync length, the load will abort.

Example:

SYNC: $0029 Bytes: 52 57 35 29 6B 74 DC B5 = track 19, sector 0

2/ The LONG-SYNC RL Header:

The second header on a RL track is actually the header for RL sector 0. All RL headers begin with a $75, and contain 7 important bytes that the RL loader needs to detect. These bytes are followed by 3 or 4 GAP BYTES that are written out as #$00's. (Any attempt to read these bytes will return a different byte value each time.) The RL header block for sector 0 (1st header block) has a SYNC LENGTH of $003C bytes, though. The RL loader will fail if this sync length is not found.

Example:

SYNC: $003C Bytes: 75 93 59 25 D6 ED 7A 4C 00 00 00 00 = sector 0

The remaining headers for sectors 1 thru the maximum have SYNC LENGTHS of $0005, and are not checked by the loader.

3/ The RL Datablock:

Each datablock begins with a $6B value and follows the header for that particular sector. Each datablock contains approximately $0255 bytes of data, which is converted into normal DATA and sent from the drive to the computer. Each datablock has a sync-length of $0005 bytes, and is not checked by the loader. Sometimes a RL sector will be blank. The datablock will then begin with a $55 byte and continue with $0254 more #$55 bytes.

Example of Full RL Datablock:

```
SYNC $0005   Bytes:  6B BB C9 24 BA FF 35 DF ..........
```

Example of Empty RL Datablock:

```
SYNC $0005   Bytes:  55 55 55 55 55 55 55 55 ..........
```

4/ The RL Bit Rate:

As far as BIT RATES and storage sizes go, RAPIDLOK formats tracks in the following manners for the following zones:

Track Zone	Bit-Rate	# of Sectors
1 - 17	$60	12
19 - 35	$40	11

5/ The RL EXTRA-SECTOR:

After all the headers and datablocks for each sector of a track are written out, a special "extra-sector" is written on the disk as part of RL's main protection scheme. The block has a SYNC LENGTH of $0014, and begins with a #$55 byte. The first byte is followed by a certain number of #$7B bytes in a row, giving the entire block a specific LENGTH. A special "de-coder" master-key block is written on track 36 of each RL disk. At the beginning BOOT of the program, RL DOS moves the disk-drive head to track 36, reads in the special key, decodes it and ends up with a list of 35 numbers. Each number is the specific length of the EXTRA SECTOR for each equivalent track! During RL file loads, if the DOS extra-sector length does not match the master-key number for that track, the DOS dies. The MASTER-KEY on track 36 is the most difficult portion of RL formats to reproduce.

Example of Extra-sector:

```
SYNC $0014   Bytes:  55 7B 7B 7B 7B 7B ..... 7B   (x amount of bytes)
```

6/ Overview of RL DOS:

Each track contains sectors 0-11 (Tracks 1-17) or sectors 0-10 (Tracks 19-35). Each "sector" is composed of a header block beginning with a $75 and is followed by a data-block beginning with a $6B (or a $55 if blank). Each RL

Track 18 on ALL RL disks is formatted in standard Commodore DOS (i.e.- 18 sectors), but it also contains the RL "extra-sector" ($55 7b 7b 7b 7b ... etc). The RL auto-boot will not load RL DOS into drive memory UNLESS this extra sector is found. It uses the 2nd byte ($7B) as a decoder for the DOS stored on sectors 18,15,12,9,6 and 3.

On all RL disks released in the past 2 years, tracks 19 thru 35 have ALWAYS been formatted in RL style. Tracks 1-17 usually vary, depending upon the program. Huge programs will RL format all these tracks, others will use combinations of standard format with RL format. Often if a game has a high-score list that is saved to disk, the RL format will leave track 1 open as standard Commodore(tm) DOS so the high-score list can be written to disk using a simple B-W or U2 command. Writing out in RL format is almost impossible! (it would take up too much disk drive memory!)

8/ In Conclusion:

As we have seen, the RL format is not standard in any way to the format that Commodore(tm) DOS is used to reading. Because of this, the only way to break the protection of titles that have the RL format is to break the seperate files from the computer memory and tie them together. This, unfortunately is beyond the scope of this manual.

We can however, give you a method of reproducing most RL protected disks. This system (developed by the Kracker Jax team) is the most effective RL copier on the market. In the next section, we will document our RL copier in detail. With our scanner, you will be able to distinguish the RL tracks from the standard tracks and even know the RL copier version. Armed with this information, you will build your own copier driver. Enjoy!

track also contains a reference header AND an extra-sector of special length that must match a "master-key." Remember, during loads, RL DOS is constantly checking the special sync lengths described above. Even the slightest mis-match from the norm will halt the program load. Thus, if your DISK DRIVE speed is slightly off from 300 RPM, you may experience difficulties in loading some RL formatted programs.

If you examine the directory sectors of track 18 on a RL disk with a track and sector editor, you will notice that after each file name is a sequence of two or three bytes. RL DOS actually uses these bytes much in the way Commodore(tm) DOS does the track and sector pointer! The actual beginning track and sector number and program length are embedded (encoded) in these bytes.

Little is known about the RL master-key on track 36. The routine that RL uses to decode it can be copied, but actually writing out the key has not yet been done!

On recent RL versions (5 and 6 to be specific), they use TRACK to TRACK alignment. What this means is that if you were on track 19 and you had just read sector 0, if you were to immediately skip the drive-head to track 20 and read the first information you encountered, you would be reading the data for sector 0 of track 20! This is a very simple explanation. Sometimes track-to-track alignment can be done with a "skew". i.e. - track 19,sector 0 matches track 20,sector 6, which in turn matches track 21, sector 12. The skew is 6.

RL DOS uses a combination of blank sectors ($55) and full sectors ($6B) on one track. This track must be perfectly aligned with the track before it. When DOS finishes reading the last sector of the first track, it bumps the drive-head to the half and half track. If the track-to-track alignment is correct, it will encounter a full RL sector, and will continue the load. If the alignment is incorrect, even off by one sector!, the drive will encounter an empty sector ($55) and the loader will then commit suicide within your drive! So even if a person could exactly duplicate two RL tracks, he would also have to get the timing within his format routine exact enough to align the tracks correctly.

An example RL Protected track:

SYNC LENGTH	BYTES	DESCRIPTION
$0029	52 55 35 29 4B 74 DC B5	track 1,0 reference header
$003C	75 93 59 25 D6 ED 7A 00	sector 0 header
$0005	6B BB C9 24 BA FF 35 DF	sector 0 datablock
$0005	75 92 59 25 D6 ED 6E 00	sector 1 header
$0005	6B De 59 24 96 7B ED F7	sector 1 datablock
.....	....	....
.....	....	....
.....	....	....
$0005	75 92 E9 25 D6 ED 65 00	sector 11 header
$0005	6B F7 D9 24 EF 4E AD DB	sector 11 datablock
$0014	55 7B 7B 7B 7B 7B 7B 7B	"extra-sector" for key

7/ Points of Interest:

< < < KRACKER JAX RAPIDLOK(tm) COPIERS > > >

RL protection has offered software publishers a very effective means of copy protecting their software. Other copy utility companies have released copiers for a title or two, but because of multiple protection schemes and the extreme difficulty in writing the copiers for those titles, they have been relatively ineffective.

After many months of research and testing, we have developed copiers for what we believe to be ALL existing versions of RL. Unlike our competitors, we have not only developed individual copiers for every version of every title we could find, but have even provided you with an extremely easy way of examining and copying ANY RL protected disk released to date (July 1987). We are confident that if you follow our instructions carefully, YOU will easily construct a copier to archive your particular version of a RL protected disk.

In order to copy a RL protected disk with our system, we must first identify it as such. Companies such as Microprose(tm), Accolade(tm), Avantage(tm), Avalon Hill(tm), and Capcom(tm) are known users of RL. Others do exist and using our system will identify them.

The heart of our system is our RL Scanner. With this scanner you can not only tell if the disk in question is in fact RL, but also the variation of tracks and which version it is. No more guessing and endless backup attempts. You are armed with EXACT information, and that information can be plugged into a Skeleton Copier to provide fast results.

Let's try one out. From the Utility Disk, LOAD and RUN the RL Scanner. When the program is loaded, insert any suspect disk into the drive and press RETURN. If it is a Rapidloked(tm) disk, you will see the red and green indicators fill the track line. Last, the version number will appear. The RL disk is made up of two completely different formats. Those tracks shown as red donuts are tracks that must be copied with a RL copier and those shown as green circles must be copied with our Nibbler. The version number shown determines the correct RL copier to use.

After writing down all RL tracks, all regular tracks and the version number, you are ready to create your own copier. Follow these easy steps.

1/ Format a work disk. File copy from the Utility disk to your work disk the following files: RLV0, RLV1, RLV2, RLV3, RLV4, RLV5, RLV6, NIBBLER, and COPIER TEMPLATE.

2/ When completed, load the BASIC file < COPIER TEMPLATE > (from your work disk directory) and LIST it out.

3/ Lines 10 and 20 are reserved for standard tracks. Lines 30 and 40 are for RL tracks. Simply type each track number, seperated by commas, on the appropriate lines. To end the sequence, type a 0 (zero). See the example below:

```
10 data 1,2,3,4,5,6,7,8,9,18,31,32,33
20 data 34,35,0
30 data 10,11,12,13,14,15,16,17,19,20
```

```
40 data 21,22,23,24,24,26,27,28,29,30,0
```

This is an example of coping standard tracks 1-9, 16, 31-35, and Rapidloked(tm) tracks 10-17, and 19-30. Notice the zeros ending the sequences in each copier type. Do not end a line with a comma, or forget to place information on EACH data line. This WILL cause the copiers to stall.

4/ Line 50 contains the title information. You may type any title you wish instead of PARAMETER TITLE. Adjust the quotes to suit the length of the title.

5/ Line 60 must contain the proper version number of RL copier to use. Type the correct number in the quotes following the RLV . Be sure to press RETURN to lock in all changes made.

6/ When these steps are done, list the parameter out again and double check the changes. If all is well, you may save the file to your work disk. Name it appropriately as it is a custom copier for your title.

To use your custom copier, simply load and run it. The screen will prompt you for disk swaps. When the procedure is done, power down and up again and try your copy. It should run just as the original did. If not, double check your parameter for possible errors.

In Conclusion:

We've found NO RL'ed titles we couldn't back-up. We HAVE had to try a different drive on occassion. Some drives just don't like writing some titles. 1571 drives seem to be extremely effective copiers using this system, but most 1541 drives will work fine. Also, we have had to copy a title or two a bit differently than normal. The tracks had to be copied in an out of order sequence.

Other points of interest are: This system only works with working ORIGINALS. Backups made with other copiers can't be backed up. You may back up second generations of backups made with this system, but you must use the RLV0 copier with the correct track sequences (again, use the scanner). The original protection scheme is flakey in loading and the copies are no better (sorry).

< < < INTRODUCTION : PROTECTION SCHEME TYPE I > > >

GEOS$^{(tm)}$ (Graphic Environment Operating System). from Berkeley Softworks$^{(tm)}$, has revolutionized the way people use their C-64s. It's icon-based, user-friendly, desktop interface has extended the life of this machine to 1990 and lured leery buyers into the world of Commodore$^{(tm)}$ computing. With the newly available 1764 RAM expansion, GEOS$^{(tm)}$ will allow a C-64$^{(tm)}$ to approach the capabilities of its younger, but more powerful brother, the C-128$^{(tm)}$.

But unlike other operating systems (CP/M$^{(tm)}$, MS-DOS$^{(tm)}$..), GEOS$^{(tm)}$ is copy-protected. Who needs a copy-protected operating system? What if you own a large selection of GEOS$^{(tm)}$ application programs and your GEOS$^{(tm)}$ original crashes? The programs are useless while you try to attain a replacement and you can't borrow a friend's copy because of the serial number protection! Clearly, it benefits only Berkeley$^{(tm)}$.

Meanwhile, we've been agonizing over which Berkeley$^{(tm)}$ releases to cover in this edition of Kracker Jax Revealed. We were reasonably sure that most of you would own GEOS v1.2$^{(tm)}$ and Deskpak I$^{(tm)}$, so we've included those. PLUS a quick-n-dirty way to defeat GEOS v1.3's$^{(tm)}$ "Trojan horse" scheme, which will erase your system files if the file "GEOSs$^{(tm)}$Boot" fails a checksum test. GEOS v1.3's$^{(tm)}$ protection might be covered in a future edition if readers demand it, but its complexity might be intimidating to some.

Be forewarned,though, that the going will be tough if you aren't familiar with "The Official GEOS Programmer's Reference Guide$^{(tm)}$" or Richard Immers/Gerald Neufeld's "Inside Commodore DOS$^{(tm)}$". GEOS$^{(tm)}$ and it's protection schemes are heavily I/O bound and good working knowledge of the 1541 drive and GEOS$^{(tm)}$KERNAL routines is essential to understanding the following articles.

Please note:
Geos, Geos v1.2, Deskpak I, Geos v1.3, Berkeley Softworks, and The Official Geos Programmer's Guide are all registered trademarks of Berkeley Softworks.

< < < HOBBLING GEOS v1.3's$^{(tm)}$TROJAN HORSE : BERKELEY SOFTWORKS$^{(tm)}$ > > >

The now infamous 'Trojan Horse', is an incredibly sneaky and rather sloppily-executed scheme that deletes your system files "GEOS"$^{(tm)}$, "GEOS BOOT"$^{(tm)}$, "KERNAL" and "DESKTOP"$^{(tm)}$ from an unauthorized copy of GEOS v1.3$^{(tm)}$ while you are rearranging your directory pages. It usually occurs within four moves. It actually doesn't delete the files, it completely zeroes out their directory entries.

The mechanism, located in "DESKTOP"$^{(tm)}$, is rather simple. A counter is incremented randomly during directory moves. At certain intervals, a checksum routine is performed on "GEOS BOOT"$^{(tm)}$. If the checksum is wrong, the Desktop$^{(tm)}$ checks the first four entries of the first directory page for GEOS$^{(tm)}$ file-type $0C (system boot file). If they match, it fills them with 00's and writes the block back to disk. The disk is no longer bootable unless you can re-create the directory entries.

The GEOS$^{(tm)}$ file-type I.D. is located in byte # 24 (18) of each file's directory entry. If this byte is changed to a GEOS$^{(tm)}$ system file-type ($04) in the above-mentioned files, the old horse never gets rolled into Troy and you can rearrange your directory with peace-of-mind.

Please note:
Geos, Geos v1.3, Desktop, and Berkeley Softworks are registered trademarks of Berkeley Softworks.

< < < PROGRAM: GEOS v1.2(tm) <> PUBLISHER: BERKELEY SOFTWORKS(tm) > > >

1/ A fast-copied or nibbled copy of GEOS v1.2(tm) will not run. It will merely do a system reset after the protection check. An error scan shows no normal DOS errors but there is data on track 36 (visible with a good GCR Editor). Track 36 is not normally copyable because it has no sync marks.

2/ Load the $C000 monitor "49152" from your Utility Disk then load "GEOS"(tm) from a backup copy of GEOS v1.2(tm). Disassemble the code at $0123. This routine loads "GEOS BOOT"(tm) and jumps to $6000. Load in "GEOS BOOT"(tm) and disassemble the code at $6000. Examination of the code reveals that the majority of it is encrypted but the decryption routine at $606C is rather simple. The code will decrypt it for us by placing a BRK instruction at $6086 and executing the code at $606C.

3/ Now look at the code again. Sharp-eyed hackers will notice the drive code starting at $623F. Here's some of the other high points of the loader:

```
        $6167 - Print "Booting GEOS(tm)...".
        $6177 - Execute Memory-Write command and output fast-loader routine to
                drive, then send Memory-Execute command at $61AD.
        $6013 - Direct I/O to drive through the serial port $DD00.
```

After the Memory-Execute command is sent, the code at $61BB waits for a signal back from the drive. At $61D4, a byte comparison is done. If it fails, the JMP instruction at $6086 is altered to $FCE2 (C-64 system reset). It then Jumps back to the decryption routine which, this time, re-encrypts the code and then performs the system reset. Let's disable the reset by placing a "BEQ $61EC" at $61D8. Re-encrypt the code by again executing the routine at $606C. Note the new encryption values at $61D8. These will be written to the proper sector on your backup copy.

4/ Load the sector editor from the Utility Disk and trace the "GEOS BOOT"(tm) file on your backup copy. Address $61D8 would be in the second block of the file (it should be Track/Sector 1/4) starting at byte # $DE (222). Place our byte changes there and rewrite the sector back to the disk. Now reboot GEOS(tm). What happens? No reset this time but the drive shuts off and the screen fills with garbage. The real meat must be in the drive code.

5/ Use the sector editor to restore T/S 1/4 back to its original state. Again load the $C000 monitor and "GEOS BOOT"(tm). Decrypt the code again as mentioned above. The drive code starts at $623F but we want to relocate to an address we can equate to the actual drive address. This code is written to $0300 in drive memory so lets move our code to $1300 (T 632F 642B 1300). The Memory-Execute command at $60CD jumps to $0375 in the drive so disassemble code at $1375. Remember to add or subtract $1000 from the address references (i.e. JSR $0300 - the subroutine would be located at $1300) when appropriate.

6/ Study the code for a while just to get a feel for it. Remember from our scan of the disk that track 36 is suspicious. 36 in hexadecimal is $24. See any references to $24? That's right! At $143A, the accumulator is loaded with the value $24 then the subroutine at $13BB ($03BB) steps the head to track $24 (36). Then a counter of $8000 (32,768) is set up, and a comparison for specific byte values read from track 36 begins. If the counter times-out to zero or all values don't match, the code at $148A is executed. Otherwise it branches to $1485. We want it to branch to $1485 unconditionally. A great place would be at the first byte comparison from $1463 - $1466: if the byte's not equal, make it go to $1485 (A 1465 BNE $1485). Apply this change to the equivalent drive code at $63A4.

7/ Re-encrypt the code again by re-executing the code at $606C. Note your encrypted byte changes and use the sector editor to write them to your backup copy. It should be Track/Sector 1/20, byte positions $AE/AF (174/175). Also make sure you have corrected the first change we made. Now reboot the GEOS$^{(tm)}$ backup. "Booting GEOS$^{(tm)}$..."... no reset... You hear the drive head swing out to 36 and back. Its loading! The screen clears, the Desktop appears, and ... where's the mouse pointer? The joystick's dead. We've been caught! But how?

8/ The most common method is through checksums. If any bytes in the code have been changed, a checksum routine will usually detect it. The protection scheme can then assume tampering and take appropriate action. We could hunt for the checksum code or we could cover our tracks. Let's try covering our tracks.

9/ We really only altered one byte in "GEOS BOOT"$^{(tm)}$ but we'll have to change a few more to pull this one off. Where could we place our code? A technique we use is to add it right to the end of the file. The last byte of "GEOS BOOT"$^{(tm)}$ is at $642B so we can start our code at $642C. But what's going to call our routine? Look for a jump instruction away from the $6000 area. At $621F, the code jumps to the $C000 area. Change that to jump to our code (JMP $642C).

10/ Now we have three bytes to correct: the drive code branch address at $63A5 and the JMP to our new code at $6220/6221. Our new code should be similiar to the following:

```
  A 642C LDX #$E7  ; restore original drive code BNE address
         STX $63A5
         LDX #$03 ; restore original JMP address - lo-byte
         STX $6220
         LDX #$C0 ; restore original JMP address - hi-byte
         STX $6221
         JMP $621F
```

Re-encrypt the code and look at our new code at $642C. It, too, has been encrypted. Write down the encrypted bytes and the new jump address at $6220. We'll write these to the backup.

11/ After loading the sector editor, write our new, encrypted jump address to Track/Sector 1/20 - byte position 40 ($28). Then add our new, encrypted code to the last sector in the file - T/S 1/7. Don't forget to change the last byte pointer at position 1 to the last byte of the new code. Using the above example code, the new bytes would be start at position 56 ($38) and the last byte would be at position 73 ($49). Position 1 will changed to 73 ($49).

12/ Now reboot GEOS$^{(tm)}$. It should load clean as a whistle. Just remember to watch your step when dealing with protection from Berkeley Softworks$^{(tm)}$. They are notorious for their endless checksum routines.

Please note:
Geos, Geos v1.2, and Berkeley Softworks are registered trademarks of Berkeley Softworks.

< < < PROGRAM: DESKPAK I(tm) <> PUBLISHER: BERKELEY SOFTWORKS(tm) > > >

Dealing with Berkeley's(tm) protected applications presents a two-fold problem: 1) The installation code, which stamps your GEOS(tm) serial number on the master and does a protection check and checksum routine. 2) The !%&%#$'&$ serial number verification that prevents you from taking your GEOS(tm) application to a friend's house and using it with his GEOS(tm). Both, however, are relatively easy to break. This will be a general discussion of the first-generation of Berkeley(tm) applications, using Deskpak I(tm) as an example.

The protection scheme on this first-generation is essentially the same. The code first checks to see if the disk has been installed. If it hasn't, it whips out to Track/Sector 35/0 and reads in the block. The block contains a direct I/O routine and some drive code that looks for non-standard data. If every thing checks out, it installs your internal GEOS(tm) serial number to the master (no write-protect tabs allowed). It never does the check again, allowing you to copy the application to work disks. From then on, it does nothing but the serial number check. This works fine in theory, but is rather inconvenient if you want to show it to somebody else and you've forgotten your copy of GEOS(tm).

The protection does checksum itself, however. To bypass this, we'll demonstrate a technique we use called the byte-swap. This entails switching bytes in the code among themselves to force the protection to pass.

Get out your GEOS Programmer's Reference Guide(tm) and make a backup of an UNINSTALLED Deskpak I(tm) master. Load the "DESKPAK READ"(tm) file from the Utility Disk and run it. The program reads Track/Sector 35/0 into 32768 ($8000) in memory. Load the $C000 monitor ("49152") from the utility disk and study the code at $8000. Look up the GEOS(tm) subroutine calls in the reference guide. Half of this code is the drive routine that is sent to the 1541. The other half suspends GEOS(tm) I/O and sends the drive routine to the 1541.

The protection check itself is at $803E. It reads in some bytes and compares them. If they all match, it falls through to $8061. Otherwise, it branches to $8064. In fact, its not unlike GEOS v1.2(tm) protection (see previous GEOS v1.2(tm) discussion). We can break the installation protection right here. However, we must contend with a checksum routine located in the main code, so we must keep the bytes intact. A simple way is the byte swap. The code contains many branch instructions. What if we swapped a BEQ (branch-if-equal) and a BNE (branch-if-not-equal) instruction at just the right place? Experimentation will reveal that swapping the branch opcodes at $803C and $804B will force the code to go to $8064.

Write this change to Track/Sector 35/0 using Disk Doctor from the Utility Disk. Load "GEOS"(tm) and boot "Graphics Grabber"(tm) (the only protected application on the disk). The protection fails. Look at the code at $8061-$8065 again. There are two sets of LDA instructions there, each loading a different value. Why not try another byte swap? Switch the two bytes that are being loaded at $8061-$8065. Now it will be forced to load a different value. Make this change to sector 0 on track 35. You should now have both sets of byte swaps written to 35/0. Boot "Graphics Grabber"(tm) again. This time it installs successfully. But you still can't use it with a different GEOS(tm), only the copy from which it was installed.

The serial number check is really the toughest part of some of the applications. Writer's Workshop$^{(tm)}$ and GEOdex$^{(tm)}$ both try to disguise the call to GetSerialNumber", an internal GEOS$^{(tm)}$ routine ($C196). One uses encryption and the other uses GEOS's$^{(tm)}$ "CallRoutine" which does an indirect JSR (Jump-To-Subroutine) to the serial number routine. An additional problem is that GEOS$^{(tm)}$ workspace starts at $0400 in memory, which the c64 normally uses as screen memory. Resetting the computer will lose all the the code located from $0400-$0800. Yet another problem is that some of the applications are stored in VLIR (variable length indexed record) files, which are split into multiple parts and special modifications have to be made to the directory to load these files like normal programs. We'll save these for a future exercise.

Deskpak I's$^{(tm)}$ serial number check is conveniently located at $2362 on our version. To catch this code, reset the computer while the application is loading. Load the "49152" monitor and dissasemble the code at $2362. You'll see this same routine in most of the Berkeley$^{(tm)}$ applications. It first checks to see if the serial number is zero. If it is, it executes the install routine that we disabled earlier (the GetBlock and checksum routine starts at $2448). If the serial number is there, it branches to $240D and checks the serial number in GEOS$^{(tm)}$ to see if it matches. If it doesn't, it displays a Dialogue Box asking you to reboot with the correct GEOS$^{(tm)}$.

The whole protection and serial number check can be disabled rather simply by placing a CLC (clear-carry-flag) and RTS (return-from-subroutine) instruction at the top of the code ($2362). On our version of Deskpak I$^{(tm)}$, the location on the original is Track/Sector 12/18, byte position # 156 ($9C). You might have to calculate the position or do a manual search of the file to track down the offending code. Write byte values 24 ($18) and 96 ($60) to the appropriate location in the file. You should have no trouble booting "Graphics Grabber"$^{(tm)}$ from any copy of GEOS$^{(tm)}$ now.

A good Snapshot type utility is helpful for some of the latest applications (GEOfile$^{(tm)}$ etc...). They will inevitably place the protection in screen memory and the snapshotter can capture that code for your casual viewing.

Please Note:
Deskpak I, Geos, GEOS Programmer's Reference Guide, Graphics Grabber, Writer's Workshop, GEOdex, GEOfile, and Berkeley Softworks are registered trademarks of Berkeley Softworks.

< < < PROGRAM: DEEP SPACE(tm) <> PUBLISHER: SIR TECH(tm) > > >

Procedure:

Loading the original produces a rattle-free load, and an error scan shows no standard errors. A backup made with the C-64 Fast Copier produces a non working backup. A backup made with a nibbler produces the same non working backup. Before starting, make a fast copy using the C-64 Fast Copier and use the Disk Logger to log the files.

Working with your backup:

1/ The disk log shows us that the boot file "DS" loads into memory from $0302 to $09EB. This means that it starts in the autoboot area and runs through screen memory and into BASIC RAM. Load the boot and you will see the screen react and the program will fail in the first few seconds. This means the protection is probably in the boot file.

2/ Turn off the computer and insert Hesmon(tm). X to Basic and load the boot file < LOAD "DS",8,1 > . When the program stalls, hit < RUNSTOP/RESTORE > to activate the monitor. Interpret memory starting at $0801 < I 0801 > because this is the beginning of BASIC RAM. Scroll down through memory and notice the BASIC program there. The Boot starts at the autostart vectors for BASIC and continues on to place a BASIC boot in memory. This is a good way to hide it from the average person. Type < X > to return to BASIC and type < LIST > to see the boot. Inspection shows that this is the protection check as well as the loader.

3/ Lets go through the code line by line.:

1- Lock up the keyboard and set number of trys to 0.
2- Intialize the drive.
3- Send Memory-Writes to the drive locations $06/$07 which represent the Track and Sector read into $0300 in the drive. The Memory-Writes place a 37/0 into those locations (Track 37/Sector 0).
4- Send Memory-Write to drive location $00 (Job Queue) $B0 (dec 176)=Seek any Sector.
5- Set up Memory Read loop of drive location $00.
6- Get value at $00.
7- Set a numerical value for E (M-R value). If trys=500 then test for protection pass.
8- If E has not been read in as an error code ($01-$10) then try all over again.
9- Initialize, close channels, and test E for $01: job completed successfully, and if so then branch to line 10 (pass protection). If not, goto line 10 and crash.
10- Jump to $02A7 and crash (because no loader has been poked in.
11- Poke in a loader and JUMP to it.

4/ Armed with this information, the way to break this code easily is to delete line 10. One way to do that is to put a REM right after the 10 which will nullify the whole line. The REM instruction is actually represented by one byte called a token. It is a 143 in decimal. We can easily install the byte with Disk Doctor.

5/ From the Utility Disk, load Disk Doctor < LOAD "DISK D*",8 > and < RUN >. At Track/Sector 18/1 Cursor to positon 3 and < j > Jump to the first sector of the DS file. Use the < n > key to follow the file to Track/Sector 31/4. Cursor to position 232 and use the < @ > key to change the Poke byte to a REM with a 143. Hit < r and y > to rewrite the sector.

6/ You'll find that the backup works perfectly now and can probably be file copied.

< < < PROGRAM: GRAPHICS INTEGRATOR II(tm) <> PUBLISHER INKWELL(tm) > > >

Procedure:

Loading the original produces a rattle-free load, and an error scan shows no standard errors. A backup made with the C-64 Fast Copier produces a non working backup. A backup made with a nibbler produces the same non working backup. Before starting, make a fast copy using the C-64 Fast Copier.

Working with your backup:

1/ Before beginning the break, let's repair the directory so we can view our files. From the Utility Disk, load the Disk Dr. as < LOAD"DISK D*",8,1 > and RUN. When the title screen comes up, insert your backup and hit RETURN. The Track/Sector brought up will be 18/1 which is the first sector of directory entries. To repair the directory, you must fill the following positions (in decimal) with shifted spaces (decimal 160).

```
pos   4-44
pos  72-76
pos 104-108
pos 136-140
pos 169-174
pos 200-204
pos 232-236
```

When your changes have been made, hit <r> for rewrite and <y> for yes. Now hit <n> for the next block (Sector 7) and make the appropriate changes to that Sector (pos 12). Again rewrite the track and hit <n> to go to the next block (sector 5). Notice the first two bytes direct the load back to Track 18/Sector 1 which causes the endless directory. Using the <@> key, change position 0 and 1 to 0 and 255 respectively. Again be sure to rewrite the Sector.

Finally with the <b> command, go back to Track 18/Sector 0. Repair the title and ID by using the <t> text command and placing spaces at position 144-148 and at pos 162 give a new ID number such as GI and again rewrite the sector. Power down and check the directory. It should appear normal.

2/ With the directory repaired, you may use the Disk Logger utility from the Utility Disk to log all files on the backup. < LOAD"DISK LOGGER",8,1 > . Inspection of the log shows a file that resides in BASIC memory starting at $0801 which is the beginning of BASIC. Let's check it out. Power down, insert your Hesmon(tm) cartridge and power up again. <X> to BASIC and load the ME file < LOAD"ME",8,1 > . List the file out. Lines 600-630 represent the call for the protection check. Let's examine the call, line by line.

600 Open channels, initialize, set the Track (T) to 34 and Sector (S) to 8.

610 Open a channel to the drive.

620 Send a Block Execute command to the drive. CHR$(66)=B CHR$(44)=- CHR$(69)=E. In other words read Track 34, Sector 8 from the disk and send it to a buffer in the drive. Execute that code starting at the first byte.

630 Close channels : RETURN to GOSUB that called the check in line 65.

3/ Let's examine the Block Execute code. From the utility disk, load the program called BLOCK READ. < LOAD"BLOCK READ",8,1 > . list the code and in line 10 set the TRack to 34 and the SEctor to 8. Place the backup in the drive and type RUN. The drive will read the proper block and transfer the code to $C000 in the computer memory. When the READY prompt comes up, hit RUNSTOP/RESTORE to enter the monitor.

4/ Begin disassembly at $C000 < D C000 > . Examine the code from $C015-$C02A. The drive reads Track 35/Sector 0 through the job Queue. The Error message is read at position $00 and if equal to $02 (header block not found), the code falls through and places a value of $7F at $003B in the drive and returns to the BASIC program that called the B-E in the first place. If the check is not satisfied, a Branch is taken to $C038 witch causes the head to go to track one and go in an endless loop.

5/ The break is now quite simple. If we place two NOPs at $C029 and $C030, the code will not be able to Branch and must fall through even if the protection doesn't pass. The changes can be made with Disk Dr. Power down and remove your Hesmon$^{(tm)}$ cartridge. Power up and with the Utility Disk in the drive, < LOAD"DISK DR",8,1 > . Use the <b> command to read in Track 34/Sector 8 from the backup. At pos $29 (decimal 41) you'll find the BNE command. Using the <@> key, change position 41 and 42 to 234 ($EA=NOP).

6/ This title is now broken and can be fast copied with any data copier. Beacuse it still uses the B-E command, you will not be able to file copy. One way to possibly break the B-E code might be to store the $7F at $3B in the drive using a M-W (memory-Write) command. Replace the B-E in the Me file with a M-W (Line 620). We will leave this to you as an exercise for further practice.

< < < MACHINE LANGUAGE MONITOR COMMANDS > > >

(public domain monitors on disk)

```
Assemble----: A aaaa ooo xxx                a  = address
Compare-----: C ssss eeee ssss              o  = opcode
Disassemble-: D ssss (eeee)                 x  = bytes
Fill--------: F ssss eeee xx               ()  = optional
Go----------: G aaaa                        s  = start address
Hunt--------: H ssss eeee xx                e  = end address
Interpret---: I ssss (eeee)                 n  = new address
Memory------: M ssss (eeee)
Registers---: R
Save--------: S "file name",08,ssss,eeee+1
Load--------: L "file name",08
Transfer----: T ssss eeee nnnn
Exit/Basic--: X
```

<==>

< < < DISK DOCTOR COMMANDS > > >

```
@ = Change Byte          t = Text Mode
+ = Scan Forward         - = Scan Back
n = Next Block           N = Previous Block
j = Jump to Link         J = Previous Link
b = New Block            B = Last Block
r = Rewrite Block        c = Copy Block
s = Swap Disks           p = Print Block
Q =Quit
```

Clear = Renew the current sector display.
Home = Position the cursor over position 0.
Cursor Keys = Position the cursor R/L or U/D.
Return = Position the cursor over the first byte of the next line.

<==>

< < < BOOKS FOR FURTHER READING > > >

Kracker Jax Revealed Vol I ....$^{(c)}$Kracker Jax Protection Busters

Commodore 1541 Disk Drive Owners Manual ....$^{(c)}$Commodore Business Machines

Commodore 64 Programmer's Reference Guide .....$^{(c)}$Commodore Business Machines

Inside Commodore Dos ....$^{(c)}$Reston Publishing Co.

Machine Language For Beginners ....$^{(c)}$Compute! Books Publication

Mapping The Commodore 64 ....$^{(c)}$Compute! Books Publication

Program Protection Manual (Vol 1 & 2) For The C-64 ......$^{(c)}$CSM Software Inc.

Official Geos Programmer's Reference Guide$^{(tm)}$......$^{(c)}$Berkeley Softworks$^{(tm)}$

< < < HES MON 64$^{(tm)}$ INSTRUCTIONS : $^{(c)}$HES > > >

****

If You've Never Used a 'Machine Language Monitor' Before

The following section is intended for people who are unfamiliar with the uses of a machine language (M.L.) monitor program. However, it is not a tutorial in the architecture of the C64 or the 6502. Nor is it intended to teach 6502 assembly language programming. In fact, some knowledge of assembler language will be most helpful. It IS intended to help the beginner get started in using HESMON. Even those who know nothing about the 6502 or the C64 will find some of HESMON's commands useful (see, for example, the Interpret Memory command).

If you are familiar with the C64's screen editor, you should have no trouble entering and editing HESMON commands. HESMON commands are entered and edited just as are BASIC direct mode commands. They consist of a single character usually followed by one or more 'parameters' and a RETURN. The parameters consist of hexadecimal numbers or character strings and are separated from one another by spaces. With one exception (the '#' command) numeric parameters must be hexadecimal and do not need to be prefixed with '$'. String parameters are identified by enclosing them in double quotes ("). If HESMON doesn't understand a command it will print '?', usually just to the right of the bad command. If the command is understood, but the result is impossible or illegal, e.g., trying to save HESMON itself on tape, HESMON prints a '?' on the following line.

To use HESMON, turn your C64 off, insert the HESMON cartridge into the expansion slot in the C64 and then turn the power on. You will see the HESMON version number, the programmer's name, the H.E.S. copyright message, and the 'cold start' register display:

```
C*
  PC   IRQ  SR  AC  XR  YR  SP
;0000 EA31  27  00  00  00  FA
```

The meaning of this rather cryptic display is as follows: The first line 'C*' identifies a cold start of HESMON, that is, starting up from power-on. The next line identifies the pseudo 6502 registers maintained by HESMON:

PC	=	program counter
IRQ	=	interrupt request vector
SR	=	status register
AC	=	accumulator
XR	=	X register
YR	=	Y register
SP	=	stack pointer

NOTE: "6502" is used synonomously for "6510" in this document.

The register contents are shown on the third line. The quantities shown in the register display (except the IRQ) are not the actual register contents, they are the numbers HESMON will use to set the 6502 registers when instructed to begin execution of a M.L. program. IRQ is not a 6502 register, but a RAM 'vector' that points to an IRQ interrupt service routine. Beginners may ignore this location — but better not change it! The ';' at the beginning of the last line is really a HESMON command. It tells HESMON (if the RETURN key is pressed with the cursor on this line) to put the seven numbers that follow into the corresponding pseudo registers. Just before beginning execution of a M.L. program HESMON copies the pseudo register contents to the 6502 registers. So, for example, if we want the C64 to print 'HI.', we could first move the cursor up to the ';' line and alter it to read:

```
 1200 EA31  27  48  49  2E  FA
```

When we press RETURN, the 6502 pseudo program counter is set to $1200; while the accumulator, and X and Y pseudo registers are set to $48 (ASCII H), $49 (ASCII I), and $2E (ASCII.). Now, if we write a program at $1200 to print the AC, XR, and YR it will print 'HI.' when we execute the HESMON Go command. Let's write such a program using the HESMON Simple Assembler command, 'A'. Type in the following lines:

```
A1200  JSR  FFD2
TXA
JSR  FFD2
TYA
JSR  FFD2
BRK
```

The 'A' beginning the first line tells HESMON we wish to assemble, that is, translate assembly mnemonics into machine code. As you press RETURN after typing each of the above lines, you will see HESMON reprint the line, showing the machine code generated from the assembly language instruction. HESMON will then prompt for the next line of program by printing the 'A' command and the next available address followed by

< < < HES MON 64(tm) INSTRUCTIONS : (c)HES > > >

a space. So you don't have to keep track of what the next address is, just type in the assembly language instructions. When you've finished the program, just press RETURN and HESMON will exit this mode. By the way, $FFD2 is one of the 'Kernal' routines in the C64's ROMs. It prints the contents of the accumulator to the current output file — the screen in this case. For further information on this and other useful ROM routines, consult the Commodore 64 Programmers' Reference Guide" published by Commodore

Now type 'G' and hit RETURN. You should see:

```
G
HI.
B*
 PC   IRQ  SR  AC  XR  YR  SP
;120C EA31 30  2E  49  2E  FA
```

Notice after the 'HI.' is another register display, the break entry display identified by 'B*'. This means we've re-entered HESMON by executing a BRK instruction — the one at the end of our short program. Now examine the register contents. The PC points one address higher than the BRK instruction. The X and Y registers and stack pointer are unchanged. The accumulator now has the $2E transferred into it by the TYA instruction at $1207. Let's play with this a bit. Type 'D1200 120B'. This command instructs HESMON to 'disassemble' the program you just entered.

Now, move the cursor to the last line, at address $120B, and type the following, with the 'A' replacing the ',' (also be sure to blank out any characters left on the screen after the '8'):

```
A120B  LDA  #48
JMP  1200
```

We now have a M.L. program that will print 'HI.' forever — or until we stop it. Type 'G1200'. When you tire of watching the stream of 'HI.HI.HI.'s, press — no, not the STOP key — the RESTORE key by itself. The RESTORE key is HESMON's super-STOP key. It will halt just about any M.L. program (except HESMON itself) when HESMON is plugged in. (Exception: If you attempt to use RS232 files all bets are off. Also, correct operation of RS232 files is not guaranteed with HESMON installed.) To get back to our example: after pressing RESTORE you should see a clear screen with the following:

```
S*
 PC   IRQ  SR  AC  XR  YR  SP
;XXXX EA31 XX  XX  XX  XX  XX
```

This is the RESTORE entry display, identified by the 'S*'. The X's are not actually what you will see. The register contents will depend upon exactly when you pressed RESTORE.

If you want to enter a series of bytes into memory, use the Memory Modify command (:). For example, to enter the sequence $01, $02, $03, $04, $05, $06, $07 . . . starting at $1234, you type:

```
:1234 01 02 03 04 05 06 07 08
```

HESMON will respond by reprinting the line and will prompt for another line by printing the next available address. As with the Assemble command, you may exit by typing RETURN.

Besides entering programs and data into memory, one of the functions of a M.L. monitor is to examine programs and data already in memory. HESMON has several commands for this purpose; including Disassembly (D), Memory Display (M), and Interpret Memory (I). These three commands are special in that the cursor-up and cursor-down key may be used to 'scroll' their displays forward and backward through memory. The action of this scrolling is easier to use than to describe. Think of the text on the screen as being on a drum which may be rolled up or down using the cursor up/down key. The scrollable display type found closest to the edge of the screen where new lines will appear is continued in the scroll direction. I said it was hard to describe! Try it. Just type 'DAAD7' and hit RETURN. Then press and hold the cursor-down key. To scroll up, go to the top of the screen and then hold down the cursor-up key.

Other commands allow you to hunt for a particular sequence of bytes in memory (H), compare two blocks of memory for differences (C), or transfer a block of memory to a different location (T). There are also two advanced functions: N—relocate absolute memory references in a program, and E—change the external references in a program. Finally, there are number base conversion and hexadecimal arithmetic functions.

< < < HES MON 64(tm) INSTRUCTIONS : (c)HES > > >

****

Alphabetical List and Description of HESMON Commands

The following section lists the HESMON commands in alphabetical order describing each in detail and giving example(s) of its usage.

A — The Simple Assembler

The HESMON simple assembler provides an easy way to enter short M.L. programs. It does not have all the features found in a complete assembler such as HESBAL in HES's 6502 Professional Development System for the VIC and Commodore 64, but it provides increased convenience compared to POKEing from BASIC or entering hexadecimal codes using a more primitive monitor. The syntax of HESMON's Assembler command is as follows:

```
A 1111 MMM OOOOO
```

where '1111' is a four digit hexadecimal address in the C64's RAM, 'MMM' is a standard three character assembler mnemonic for a M.L. operation code (op-code), such as JSR, LDA, etc. 'OOOOO' is the 'operand' of the op-code. It is beyond our scope here to discuss fully the meaning of those parameters — for a complete discussion, consult a book on 6502 assembly language programming. See Section I for a simple example of A's usage. Notice that since all numeric operands MUST be in hexadecimal notation the customary '$' preceding these numbers is optional; as is the ',' preceding 'X' or 'Y' in indexed instruction operands. If HESMON understands the line, it will reprint it showing the corresponding byte(s) of M.L. between the address and the assembly code. HESMON will then prompt for the next line of assembly code by displaying the next address followed by a space and the input cursor. If HESMON cannot interpret the line, it will print a '?' instead of prompting for the next line. For example, you type:

```
A 1200 LDA #41
```

HESMON responds by overprinting your line and then prompting for the next line as follows:

```
A 1200 A9 41     LDA #$41
A 1202
```

Note — HESMON ignores anything to the right of a ';' on the line.

B — Breakpoint Set

There are three different methods to return to HESMON from a M.L. program. The Breakpoint Set command is one of them. This command allows you to designate an address in a program as a 'breakpoint,' that is, a place where the program is to be halted and control is to be returned to HESMON. Breakpoint Set also allows you to specify the number of times the instruction at this address is to be executed before the breakpoint is activated. The breakpoint defined with Breakpoint Set is effective ONLY when the C64 is executing HESMON's Quick Trace command. For example, to halt a program, that starts at address $1200, on the fifth repetition of the instruction at address $1234, you would type:

```
B 1234 0005
Q 1200
```

The first line above sets the breakpoint at $1234 and the repeat count to five. The second line initiates the Quick Trace mode of program execution (see the Quick Trace command). When address $1234 has been reached for the fifth time HESMON will halt execution of the program, display the current values of the 6502 registers, and enter the single-step mode of execution (see the Walk command).

The second method to return to HESMON from a M.L. program is to insert a 6502 'BRK' instruction into the program. Obviously, since this method requires program modification, it may be used only with programs in RAM. Finally, HESMON may be called by simply pressing the RESTORE key. In either of these last two cases HESMON will be re-entered whether or not the Quick Trace mode was active. If a BRK instruction was encountered, the 'break' entry register display will be printed showing the contents of the 6502 registers. Similarly, if the RESTORE key is pressed, the RESTORE entry register display is shown. In the latter case, the screen is cleared first. The RESTORE key method of HESMON re-entry will work any time the HESMON cartridge is plugged in — unless an RS232 file has been accessed or the 6502 has attempted to execute an undefined op-code (one that disassembles as '???'). After an RS232 file has been attempted HESMON may be re-entered from BASIC via a BRK instruction. Type 'SYS8' to cause a break entry.

< < < HES MON 64(tm) INSTRUCTIONS : (c)HES > > >

****

C — Compare Memory Blocks

This command compares two sections of memory and reports any differences by printing the address of one member of the mismatched pair(s). The syntax is as follows:

C 1111 2222 3333

where 1111 is the start address of the first section, 2222 is the end address of the first section, and 3333 is the start address of the second section — the one to be compared with the first section. This command may be stopped (in case a large number of addresses are printed) with the STOP key. For example, suppose you have two disk files containing (you thought) the same M.L. program residing at locations $1400 to $147F. However, when you used the BASIC command VERIFY, it said 'VERIFY ERROR'. Naturally, you wonder just where the difference is. VERIFY can only tell you they differ SOMEWHERE. Compare Memory Blocks may be used to find out: First use HESMON's Load command to load one of the files (See Load). Then move that program to $1500 using the HESMON Transfer Memory Block command: T 1400 147F 1500. Next Load the other file. Now compare the two files using Compare Memory Block:

C 1400 147F 1500

HESMON will print a list of all the memory locations which differ between the two programs.

D — Disassemble Memory

This command is the inverse of the Assemble command. It interprets memory contents as M.L. instructions and displays the assembly language equivalent. Disassemble is used in two distinct ways. First, it may be used to disassemble a section of memory by specifying an address range, such as:

D 1111 2222

where 1111 is the start address and 2222 is the end. This type of disassembly is convenient when used in conjunction with HESMON's Output Divert command to produce a hardcopy listing of a M.L. program. Second, the disassemble command may be started by entering a single parameter, the beginning address:

D 1111

This mode is handy for examining a M.L. program on the screen because, once the first line is displayed, preceding or subsequent lines of code may be disassembled by pressing the cursor-up or cursor-down key respectively.

You may alter a program in RAM using the Disassemble command's output. If you move the cursor to the line you wish to alter, change the byte display (not the mnemonic), and press return; HESMON will alter the memory contents and retype the line showing the altered bytes and the corresponding disassembly. Then HESMON will prompt for the next line by printing the next address and leaving the input cursor on the same line. To exit this mode type RETURN, just as with the Simple Assembler command.

E — External Relinker

This command is rather difficult to understand, but the effort is worth it! Basically, this command facilitates the transport of M.L. programs from one 6502-based computer to another (PET, VIC, etc.) by translating the system calls of one computer to those of another. Of course the capabilities of these computers are different so one cannot always achieve a perfect translation, but at least a functioning version can be made without completely rewriting the program. The heart of this command is a table of corresponding addresses. This table contains four-byte entries consisting of pairs of addresses. These address pairs are the addresses in the respective computer operating systems that perform a given task. Typically these will be addresses in the ROM firmware of the computers. The correspondence table must be supplied by you. Lists of common ROM routine addresses in various 6502 computers have appeared in several places, most notably in COMPUTE! magazine (e.g., "VIC Memory Map Above Page Zero", COMPUTE! Vol. 4, No. 1, P. 181); "Butterfield on Commodore", Commodore Magazine, Oct./Nov., 1982, pp. 81 ff.; and, for the PET, in "PET/CBM Personal Computer Guide" by Osborne and Donahue.

For example, suppose you have loaded into your C64 a M.L. program intended to run in a PET with BASIC 4.0 ROMs. We will assume it is in locations $1200 to $13FF. Many of its external subroutine calls are probably of the form JSR $FFxx. The subroutines at these addresses are all almost identical in function to those of the same address in the C64 because these entry points are in a 'jump table' set up for the purpose of standardizing system calls between the different Commodore ROM sets. So what's the

< < < HES MON 64$^{(tm)}$ INSTRUCTIONS : $^{(c)}$HES > > >

****

problem? Any subroutine call in the address range $B000 to $FF00 probably also has an equivalent in the VIC, but it's at a different address! This is where External Relinker comes in. Extemal Relinker will find such subroutine calls and replace them with the corresponding C64 ROM routine calls — if we can identify the correct replacement (this is where the published ROM maps come in). If we already have a correspondence table constructed in an earlier session with Extemal Relinker, we simply load it using the Load command. But, if we don't have a table, External Relinker will use our answers to its queries to construct one we may save for future use. For the present example, suppose we have no table, just two ROM maps. We want to construct a table starting at $1000, so we start it by entering four zeroes (four zeroes denote the last entry in the table) using the Fill Memory Block command.

F 1000 1003 00

Then we start External Relinker:

E 1200 13FF 1000 B000 FF00

The first two parameters tell External Relinker where the start and end of the program we are working on are. The third says where the correspondence table starts. The last two give the address range we're interested in relinking. At this point Extemal Relinker will start disassembling our program from $1200 to $13FF, looking for references to addresses in the specified range of $B000 to $FF00. When it finds such an address it will first consult the correspondence table which starts at $1000 — if no entry for the address is found, it will show the disassembled line containing the unknown address and wait for the entry of the correspondence address. We will look up the PET address in the published table, find its equivalent in the C64 table, type the VIC address over the one on the screen, and press RETURN. HESMON will add the new correspondence to its table, alter the address reference in the program and then continue its search. On subsequent occurrences of this address HESMON will automatically make the specified replacement.

F — Fill Memory Block

This command is used to set a section of memory to a particular value. The syntax is as follows:

F 1111 2222 33

where 1111 and 2222 are the first and last addresses (inclusive) of the section to be filled and 33 is the hexadecimal quantity to be written. See, for example, the usage in the example of Extemal Relinker.

G — Go (execute program)

This command transfers control of the C64 to a M.L. program; that is, it starts execution of the M.L. program. It may be used with or without an address parameter. If no address parameter is given, execution is begun at the address shown in the program counter (PC) of the Register Display command. For example you may exit HESMON and 'warm start' BASIC by typing:

G A474

The C64 will respond, "READY.". For another example, see Section 1.

H — Hunt for a Sequence

This command locates a specific sequence of bytes in memory. It has two forms, as follow:

H 1111 2222 33 44 55 . . . .
H 1111 2222 "ABCDE . . . . "

where 1111, 2222 are the first and last addresses of the range of memory to be searched and 33, 44, etc., are the hexadecimal byte(s) to be found, separated by spaces. The second form allows the bytes to be specified as characters enclosed by quotes. For example to find all subroutine calls to the character output routine (AB47) in the C64 ROM's we would type:

H A000 FFFF 20 47 AB

HESMON responds with a list of all such subroutine calls. Note that, as usual, the low and then high order bytes of the address were specified.

To find all occurrences of the string 'READY' (there is only one, at $A378), we would type:

H A000 FFFF "READY"

I — Interpret Memory

This command displays the contents of memory as 'ASCII' characters. It is similar to the Memory Display command except that it shows 32 characters per line. It may be used with either one or two parameters and its output may be scrolled just as with the Disassemble command. For example, to see the table of BASIC's keywords and error messages, type:

I A000 A300

< < < HES MON 64$^{(tm)}$ INSTRUCTIONS : $^{(c)}$HES > > >

****

L — Load 'Program'

This command 'loads' (i.e., reads) a 'program' into memory from an external device such as tape or disk. The loaded material need not actually be a program. For example, it may be a section of memory containing a data table for External Relinker that was saved to tape or disk using the Save command. However, the most common use of Load is to retrieve M.L. programs from tape or disk. Note that HESMON's Load should NOT normally be used to load a BASIC program. The syntax of Load is as follows:

L "programname" 11

where 'programname' is the name of the file to be loaded (be sure to include the double quote marks) and '11' is the device number from which to load. If the device number is omitted, the tape drive will be assumed; if the filename is also omitted, the first file found on the tape will be loaded. For example:

L "YAHTZEE" 08

The above loads YAHTZEE from device eight, the disk drive.

M — Memory Display

This command displays the contents of memory in hexadecimal notation. It This command displays the contents of memory in hexadecimal notation. It is similar to the Disassemble command in that it may take either one or two addresses as parameters. The two-parameter form displays from the first address to the second; the one-parameter form shows eight bytes beginning with the address given. Also like the Disassemble command, the output of Memory Display may be scrolled up or down with the cursor-up and cursor-down key. For example:

M A000 A040

shows from $A000 through $A047 in hex and in characters, eight bytes per line. To see more, press cursor-up or -down.

N — New Locator

This command is a relative of the Extemal Relinker command. It has a different general purpose, however. New Locator is designed to convert absolute address references in a M.L. program from one memory range to another. It is typically used following a Transfer Memory Block command to relocate a program in memory. For example, suppose you have just moved a M.L. program from $1200-$1280 to $1300-$1380 using T. Any address references within the program now point $0100 too low. New Locator can fix this. Type:

N 1300 1380 0100 1200 1280

The meaning of the above line is as follows: Disassemble from $1300 to $1380 checking for addresses in the range $1200 to $1280. Add $0100 to any such addresses. If we had moved a table of addresses, for example a 'jump table' (pairs of numbers of addresses, low byte followed high byte), instead of actual machine code; we would put a 'W' following the last parameter to tell New Locator to treat the memory contents as pairs of address bytes rather than M.L. The general Syntax for New Locator is the following:

N 1111 2222 3333 4444 5555 [W]

where 1111 and 2222 specify the actual memory range to scan, 3333 is the 'offset' to add to adjusted addresses, 4444 and 5555 specify the address range of references which are to be adjusted, and W (if present) specifies that the scanned range is a table of 'words' with no op-codes. If not in the 'word table' mode, New Locator will halt and display any line of machine code it can't disassemble.

O — Output Divert

This command is HESMON's equivalent to BASIC's CMD command. It allows HESMON's output to be printed on the C64 printer or stored in a disk file instead of being displayed on the screen. This is the preferred method to get HESMON's output on a device other than the screen. Output Divert has a number of options. The complete syntax of the command is:

O11 22 "filename"

where '11' is the device address where the output is to be sent (normally 04 for the printer), '22' is the 'secondary address' of the device (typically 02 to 0E for the disk drive), and 'filename' is the filename to be used for storing the output (see your disk drive documentation). All of these parameters are optional. If you merely type 'O' HESMON will open a file to device 4, the printer, and start diverting its output. If you type 'O' when the output is already being diverted, the file will be closed and the output will be directed to the screen again. That is, typing 'O' 'toggles' Output Divert on and off. If you want explicitly to revert to screen output, type 'O3F'. The secondary address and filename default to 'none' since they are not needed by the printer. For more information about

< < < HES MON 64$^{(tm)}$ INSTRUCTIONS : $^{(c)}$HES > > >

****

filenames and secondary addresses, consult the documentation for the device to which you wish to divert HESMON's output.

P — Print Screen

This command is a limited version of Output Divert. It copies the current screen display to printer or disk. It's just like having a snapshot of the current screen image. The parameters of Print Screen are the same as for Output Divert, except there is no toggling because Print Screen automatically reverts to screen output at the completion of the screen copy. Note: Print Screen will NOT copy high resolution graphics.

Q — Quick Trace

This command is used after the Breakpoint Set command in debugging M.L. programs. It takes one or zero parameters just like the Go command. If specified, the parameter gives the address at which to begin execution. If omitted, execution begins at the PC shown in the register display. The difference between Quick Trace and Go is that a breakpoint, defined with the Breakpoint Set command, is only recognized in the Quick Trace mode of execution — the breakpoint will be ignored if execution is begun with the Go command. Program execution is much slower with Quick Trace than with Go because Quick Trace is really just a fast version of the Walk (single step) command. Using Quick Trace, instructions are executed one at a time and HESMON is re-entered after each. This process continues until the defined breakpoint is reached. For an example of Quick Trace usage, see the Breakpoint Set command.

R — Register Display

This command displays HESMON's current 6502 pseudo register contents as well as the current interrupt request (IRQ) RAM vector. The IRQ vector is shown as a convenience to the programmer who wishes to use this vector to run interrupt-driven or 'background' routines. This vector may be altered like any of the register contents; however, extreme caution must be exercised in so doing because the replacement is made *IMMEDIATELY*, not at the time of execution of a Go command. Therefore, the interrupt handling routine must be in-place BEFORE the IRQ vector is altered.

There are no parameters for the Register Display command, just type 'R'. To alter the register contents, move the cursor to the line beginning with ';' and overwrite the display. Then hit RETURN and the contents will be altered. Note that the display, except as noted for the IRQ vector, shows the contents of the 6502 registers at the time HESMON was entered. These registers will be set by HESMON to the values shown in the register display just prior to beginning execution of a program using the Go, Quick Trace, or Walk commands. For a fuller discussion of the meaning of this display, see Section I.

S — Save 'Program'

This command saves the contents of a specified range of memory to an external device as a non-relocating 'program' file. The 'non-relocating' part means that the program may be reloaded from tape using BASIC's LOAD command. The syntax of Save is as follows:

S "filename" 11 2222 3333

where 'filename' is the filename to be used (don't forget the double quote marks), '11' is the device number on which to save (01 for the tape and 08 for the disk drive). '2222' is the beginning address. '3333' is the last address PLUS ONE of the memory area to be saved. All the parameters must be given, except that in tape saves the 'filename' may be null (""). For example, to save a M.L. program residing from $1500 to $1DFF to the disk as 'APROGRAM', type:

S "A PROGRAM" 08 1500 1E00

Again, notice the last parameter is *one byte higher* than the last program address. Also, note that HESMON's Save should NOT be used to save BASIC programs because HESMON saves programs as absolute, not relocatable, files.

T — Transfer Memory Block

This command transfers the contents of a block of memory to another area. Its syntax is as follows:

T 1111 2222 3333

where 1111, 2222 are the first and last address (not last-plus-one) of the block to move and 3333 is the starting address where the block is to be moved to.

U — (Test Color RAM)

U has no parameters. It tests the color RAM for proper function and prints 'OK' if they are working. If there is a bad byte, its address will be printed.

< < < HES MON 64(tm) INSTRUCTIONS : (c)HES > > >

****

V — Verify RAM Function

This command tests a section of RAM for proper function. Its syntax is:

V 1111 2222

where 1111, 2222 are the first and last memory locations of the block to test. HESMON will keep cycling the test over the address range specified until the STOP key is pressed (it may be necessary to hold it down for a second or two). At the successful completion of each test of the memory block, HESMON will print a '.' to show it is working. If a memory location fails the test, HESMON will print its address followed by a binary number showing the data incorrectly stored. The bits of the number are shown most significant (bit 7) to least significant (bit 0) left to right. The bits of the RAM location that are different from the test data are printed in reverse field. Using the information printed on the screen it will usually be possible to pinpoint the bad RAM IC(s). Note that if you 'test' addresses that contain no RAM, a seemingly random pattem of numbers will be printed.

W — Walk Program

This command causes single-step execution of a M.L. program under user control. It, like Go and Quick Trace, may be used without a parameter to begin at the register display 'PC' location; or it can accept one parameter that specifies the starting address. To exit the Walk mode, press the STOP key. To step as rapidly as the registers can be printed, press the SPACE bar. To step at the key repeat rate, press a normally repeating key, e.g., the cursor down key. To take one step only, press a normally non-repeating key, e.g., the left-arrow key. The 'J' key has a special function in Walk mode. It causes HESMON to continue execution at full speed until a return-from-subroutine instruction is executed. For example, type:

W AAD7

HESMON will begin execution at $AAD7 — the carriage return, linefeed output ROM routine. After executing the instruction at that address HESMON will halt, showing the register contents and a disassembly of the next instruction the C64 will execute if Walk is continued. The display in the above example is as follows:

```
25 0D 00 00 FA
,AAD9 20 47 AB JSR $AB47
```

The first of the two lines above shows the 6502 register contents in the same order as the Register Display command: SR AC XR YR SP. This example assumes HESMON has just been cold started, otherwise the registers — except the accumulator — may differ from those shown here. The second line shows that the C64 will next do a subroutine call to $AB47, the character output routine used by BASIC. To continue, press any key except STOP or 'J' (no need to hit RETURN). Suppose we press the left-arrow key once. HESMON will now show two more lines:

```
25 OD 00 00 F8
,AB47 20 0C E1 JSR $E10C
```

Now we see the C64 is at location $AB47 about to execute a subroutine call to $E10C. Notice the stack pointer (SP) has been decremented by two because the return address for the JSR instruction was 'pushed' on the stack before the jump to $AB47 was executed. Let's press the left-arrow once more:

```
25 OD 00 00 F6
,E10C 20 D2 FF JSR $FFD2
```

Here we finally get to a place where the C64 is going to a 'Kernal' routine we can recognize: the character output routine $FFD2. Since this routine is documented in the C64 literature, we know exactly what it will do: print the character $0D in the accumulator. Therefore, we needn't single step further through that routine. So we press the 'J' key. HESMON shows (after a blank line — where the carriage return was printed):

```
20 0D 00 06 F6
,E10F B0 E8    BCS $E0F9
```

Now the C64 is at the point just following the JSR $FFD2 instruction. The 'carry' bit (bit 0) of the status register (SR = $20) is clear (0), so the branch on carry set (BCS) will not be taken. At this point we may continue to single step through this subroutine by pressing left-arrow; return to the next higher level of code (SP = $F8) by pressing 'J'; or quit the Walk command by pressing STOP.

X -Exit to BASIC

This command gives control to the C64's BASIC interpreter. It has two forms. The first form 'XC' has the same effect as if the C64 were turned off and then back on without the HESMON cartridge plugged in except that HESMON may be entered by pressing RESTORE. The second form 'X' causes a 'warm start' of BASIC,

< < < HES MON 64$^{(tm)}$ INSTRUCTIONS : $^{(c)}$HES > > >

****

similar to pressing RESTORE when HESMON is not plugged in. **Your first exit to BASIC from HESMON after turning on the C64 should be an 'XC', otherwise BASIC may misbehave.** While in BASIC, to achieve the same effect as pressing STOP & RESTORE without HESMON: First press RESTORE. Then type 'X' and hit RETURN.

# — Convert Decimal to Hexadecimal

This command prints the hexadecimal equivalent of a decimal number. If the decimal number is negative it shows the two's complement 16-bit hex equivalent and the corresponding positive decimal number. For example:

# 1234

HESMON shows (on the same line):

# 1234 = $04D2 1234

$ — Convert Hexadecimal to Decimal

This command prints the decimal equivalent of a hexadecimal number. For example:

$ ABCD

HESMON shows (on the same line):

$ ABCD 43981

+ — Hexadecimal Addition

This command prints the sum of two hexadecimal numbers in hex and decimal. All four digits, including leading zeroes if needed, must be used. Example:

+ 1234 5678

HESMON shows (beginning on the same line):

+ 1234 5678 = $68AC 26796

– — Hexadecimal Subtraction

This command prints the difference of two hexadecimal numbers in hex and decimal:

– 1234 5678

HESMON shows (beginning on the same line):

– 1234 5678 = $BBBC 48060

Notice that the decimal number in this example is positive even though we would expect the result of this subtraction to be negative. This is because the two-byte number $BBBC doesn't retain the information that the result is negative. If you want to know the true negative decimal result, either type in the operands in the reverse order, or type:

– 0000 BBBC = $4444 17476

So, the true decimal value of the difference $1234 – $5678 is – 17476.

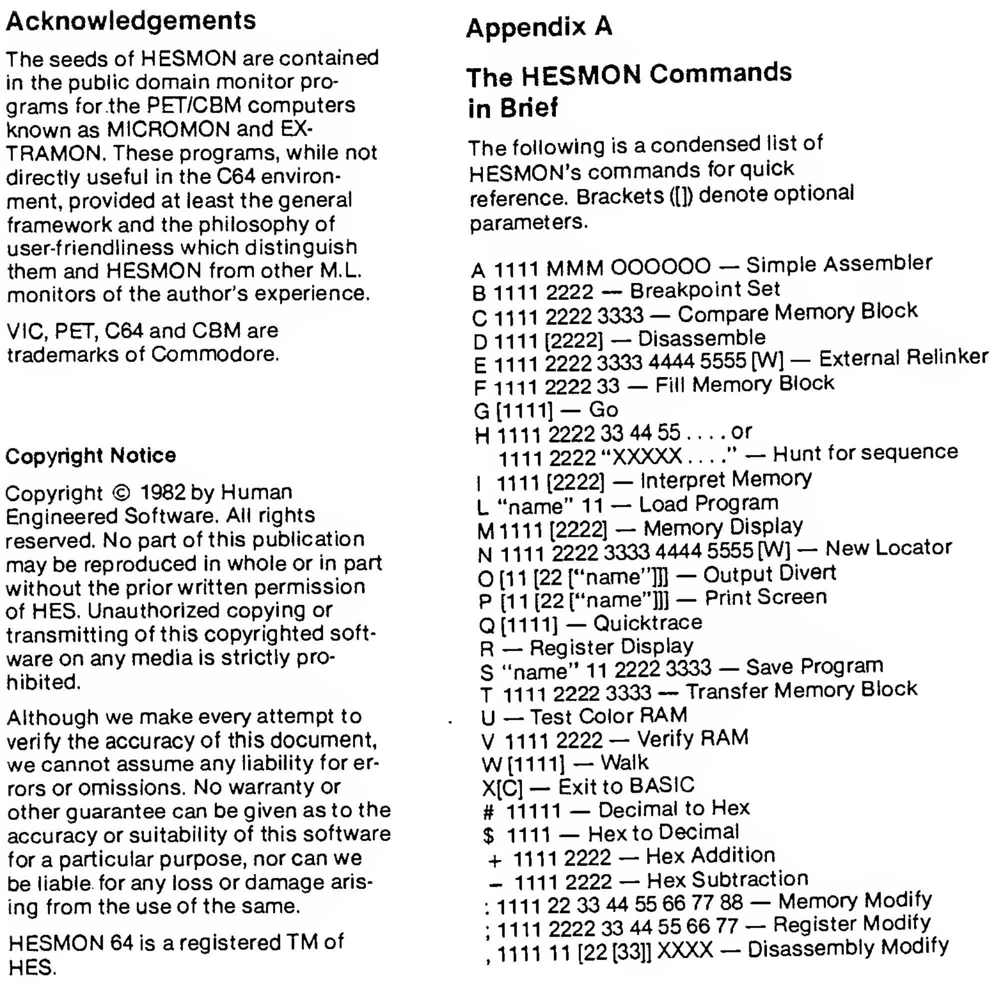
Things to be careful about when using HESMON

The BASIC interpreter has control of the C64 at all times when BASIC is running. This means that the worst that's likely to happen if your BASIC program has an error is that BASIC will issue a 'SYNTAX ERROR' message and stop your program. A M.L. monitor, on the other hand, must allow its user to take complete control of the C64 to execute certain commands. So, if your M.L. program has an error and you attempt to execute it using the Go command, the likely result is that the C64 will go catatonic — that is, even the RESTORE key may not bring back HESMON. In this event you will have to turn the power off and back on to get back to HESMON. You may avoid this catastrophe by using the Walk command to check out your program. Nevertheless, you can still send the C64 to never-never land by attempting to Walk through an instruction that disassembles as '???'. These instructions are 'unimplemented op-codes'. They do not have a defined result. Many of them cause the 6502 to 'crash' — that is, enter a state from which it may be recovered only by powering on again.

HESMON uses 33 bytes near the bottom of the machine stack ($120-$141) for its variable storage. Most M.L. programs do not use a sufficiently large amount of the stack to interfere with this storage — but it is a possibility to be aware of. Large, complex BASIC programs sometimes do use enough of the stack to interfere with these locations. And finally, RS 232 files will **not** work correctly when HESMON is plugged in.

< < < HES MON 64(tm) INSTRUCTIONS : (c)HES > > >

****



Acknowledgements

The seeds of HESMON are contained in the public domain monitor programs for the PET/CBM computers known as MICROMON and EXTRAMON. These programs, while not directly useful in the C64 environment, provided at least the general framework and the philosophy of user-friendliness which distinguish them and HESMON from other M.L. monitors of the author's experience.

VIC, PET, C64 and CBM are trademarks of Commodore.

Copyright Notice

Although we make every attempt to verify the accuracy of this document, we cannot assume any liability for errors or omissions. No warranty or other guarantee can be given as to the accuracy or suitability of this software for a particular purpose, nor can we be liable for any loss or damage arising from the use of the same.

HESMON 64 is a registered TM of HES.

Appendix A

The HESMON Commands in Brief

The following is a condensed list of HESMON's commands for quick reference. Brackets ([]) denote optional parameters.

```
A 1111 MMM OOOOOO — Simple Assembler
B 1111 2222 — Breakpoint Set
C 1111 2222 3333 — Compare Memory Block
D 1111 [2222] — Disassemble
E 1111 2222 3333 4444 5555 [W] — External Relinker
F 1111 2222 33 — Fill Memory Block
G [1111] — Go
H 1111 2222 33 44 55 . . . . or
   1111 2222 "XXXXX . . . ." — Hunt for sequence
I 1111 [2222] — Interpret Memory
L "name" 11 — Load Program
M 1111 [2222] — Memory Display
N 1111 2222 3333 4444 5555 [W] — New Locator
O [11 [22 ["name"]]] — Output Divert
P [11 [22 ["name"]]] — Print Screen
Q [1111] — Quicktrace
R — Register Display
S "name" 11 2222 3333 — Save Program
T 1111 2222 3333 — Transfer Memory Block
U — Test Color RAM
V 1111 2222 — Verify RAM
W [1111] — Walk
X[C] — Exit to BASIC
# 11111 — Decimal to Hex
$ 1111 — Hex to Decimal
+ 1111 2222 — Hex Addition
- 1111 2222 — Hex Subtraction
: 1111 22 33 44 55 66 77 88 — Memory Modify
; 1111 2222 33 44 55 66 77 — Register Modify
, 1111 11 [22 [33]] XXXX — Disassembly Modify
```

< < < EOR TRUTH TABLE > > >

****

EOR	*	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	*	EOR
0	*	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	*	0
1	*	1	0	3	2	5	4	7	6	9	8	B	A	D	C	F	E	*	1
2	*	2	3	0	1	6	7	4	5	A	B	8	9	E	F	C	D	*	2
3	*	3	2	1	0	7	6	5	4	B	A	9	8	F	E	D	C	*	3
4	*	4	5	6	7	0	1	2	3	C	D	E	F	8	9	A	B	*	4
5	*	5	4	7	6	1	0	3	2	D	C	F	E	9	8	B	A	*	5
6	*	6	7	4	5	2	3	0	1	E	F	C	D	A	B	8	9	*	6
7	*	7	6	5	4	3	2	1	0	F	E	D	C	B	A	9	8	*	7
8	*	8	9	A	B	C	D	E	F	0	1	2	3	4	5	6	7	*	8
9	*	9	8	B	A	D	C	F	E	1	0	3	2	5	4	7	6	*	9
A	*	A	B	8	9	E	F	C	D	2	3	0	1	6	7	4	5	*	A
B	*	B	A	9	8	F	E	D	C	3	2	1	0	7	6	5	4	*	B
C	*	C	D	E	F	8	9	A	B	4	5	6	7	0	1	2	3	*	C
D	*	D	C	F	E	9	8	B	A	5	4	7	6	1	0	3	2	*	D
E	*	E	F	C	D	A	B	8	9	6	7	4	5	2	3	0	1	*	E
F	*	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0	*	F
EOR	*	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	*	EOR

To find the result of an Exclusive-Or operand, locate the intersection of the 2 operands in the chart.

Example:

To find the result of $EA EOR $12:

1) The first two values to EOR are 'E' and '1'. Locate 'E' on the left or right side of the chart. Move horizontally until you intersect the column for '1'. You should find an 'F'. This is the High-Order result.
2) Repeat the process for 'A' and '2'. You should find that the answer is '8'. This is the Low-Order result.
3) $EA EOR $12 =$F8.

< < < KRACKER JAX REVEALED VOL II : (c)1987 KJPB > > >

Thank you for your purchase of Kracker Jax Revealed Vol II. We hope that this manual will provide you with information that will be useful. As you probably know, disk protection is becoming increasingly difficult to overcome with copiers. Making backups of your software will (we believe) become almost impossible unless you, 1) break the protection yourself by hand or 2) utilize special hardware. We do have plans in the future to release hardware copiers. These as all copiers, will be limited, and will have to be supported by parameters. Again, at least partial hand breaking will be required. Now is the time to understand protection and keep up with the ever changing routines.

I'd like to take this moment to thank two programmers for their part in Kracker Jax Revealed Vol II. Mike Howard, for his extensive expose' on Rapid Lok(tm)and for his deprotection methods of Access(tm) protection. Bob Mills for his efforts on the Geos(Tm) section, Aliens(tm) and Transformers(tm). As we have mentioned before, Kracker Jax is the work of a variety of programmers, each lending his expertise in his particular field. In this way, we can assure that you will be provided with the finest archival products on the market.

< < < LIMITED WARRANTY > > >

We have attemped to assure that this manual, along with the software and hardware, works as specified. We would appreciate receiving notice of any errors you may find. Neither the author nor any distributor of this product will be liable for any damages which may be a result of errors or omissions, use or misuse of this product. Should there be any defects in the software provided in this package, we will replace the defective diskette within 90 days from the date of purchase. You will find our software unprotected, and are encouraged to make a back up for your own use.

< K.J.P.B. : P.O.Box 6216 : Vancouver Wa 98668 : (206) 696-4956 >

<=======================================>

Important Notice: This instructional material has been written for educational and archival purposes only. You are advised that the Federal Copyright Law allows you the right to back up, for archival purposes, any computer program you have purchased. Any other use could be unlawful and is not advised nor encouraged. By using this product, you agree to be bound by the terms of this notice.

THE FOLLOWING TITLES ARE REGISTERED TRADEMARKS OF THE FIRMS LISTED BELOW.

Rogue Trooper	UXB.
Trio	Softsync.
Express Raider, Breakthrough	Dataeast.
Titanic, Rocky Horror Show., Aliens, Transformers	Activision.
Artist 64	Wigmore.
Colossus Chess, Falklands 82	Firebird.
Computer Scrabble	Leisure Genius.
Leaderboard, Exec.Leaderboard #1, Leaderboard Tourn.#1, Tenth Frame	Access.
Deep Space	Sir Tech.
Graphics Intergrator 2	Inkwell.
Geos v1.2, Geos v1.3, Deskpak 1, Graphics Grabber	Berkeley Softworks.

< < < PRODUCTS OR SERVICES FOR YOUR CONSIDERATION > > >

The Hacker's Utility Kit

The Hacker's Utility Kit is the finest set of disk manipulation and examination tools ever assembled into one package for the C-64. A bold claim? Maybe, but further examination of the tools available on this package will prove this. We would like to take this opportunity to go over the different utilities this package offers. The programmers who write the Kracker Jax parameters are using and enjoying these utilities that are now available to you, our valued customers.

1. Sector and Usage Scanner:
This utility will allow FAST disk scans for actual sector usage (not just BAM reads) and will report all DOS errors encountered during the scan. Programable start and ending tracks as well as printer outputs are included.

2. Density Scanner:
As in the Sector and Usage scanner, FAST disk reads, programable tracks, and printer outputs are included. Find altered densities fast.

3. Kracker Hacker GCR Editor:
This is, in our opinion, the finest GCR editor ever developed for the C-64 . So many features are included we can't begin to list them all. Some are: Edit tracks 1-40, No special sync information needed to access reads, control drive speed during reads and writes, built in help screens, note pad feature allows you to save a header in memory for comparison, and printer outputs for easy to follow hardcopies. Use this GCR Editor and you'll trash all the rest!

4. Fast Data Copier:
One or two drives. This copier skips tracks with no data for extra speed. Finished copies are cleaned of all physical protection.

5. Nibble Copier:
One or two drives. This copier again skips tracks with no data. Speed is of the essence.

6. File Track and Sector Linker:
The only one of it's kind. This utility allows you to identify the position of any sector in a file on the disk FAST. Incremented addressing, disassembly of any sector, with rewrite option, as well as a hex display available at the press of a key make this a MUST HAVE for any "Hacker". You'll love this utility.

7. Byte Pattern Finder:
Again, the only one of it's kind. We created this utility out of desperation. Finds any pattern of hex, decimal, or ASCII (even in combination with each other) you wish. Fast loaders installed for blinding speed. This is a utility you can't appreciate enough until you've used it!

8. Kracker Jax Parameter Copier Creator:
This utility was designed with you the user in mind. Now that you've determined the correct byte changes necessary to break a title, how do you implement the changes easily. The Parameter creator allows you to combine your byte changes along with a fast data copier or nibbler to create a finished copier custom built by you. The copier is then saved to any blank disk as a self running program with any title you wish to use. This utility is HOT!

9. Kracker Mon :
Created by one of our top programmers, Kracker Mon is one of the finest monitors ever created for the C-64. Features include: relocatable (per every $1000) monitors--save to any work disk. Custom change the opcode file with our user friendly editor, output to printer routines for those valuable hardcopies, and drivemon for work that has to be done in the drive memory. All monitors feature scrolling back and forth through memory, and commands are explained in the documentation to allow you to view memory in such areas as under ROMs. We feel this is a disk based monitor you will come to depend on time and time again.

10. Fast Formatter:
Take your pick : Single tracks or whole disk. This formatter is FAST (only 9 seconds MAX).

11. Disk File Logger:
This utility allows individual files or the whole disk to be logged for beginning and ending addresses. Fast loaders and printer outputs are featured. No more endless waits for the disk to log.

Well, if all this sounds like an advertisement, I guess it is (sort of). The purchase of the Hacker's Utility Kit will do two things. First and foremost, you will own an outstanding program developed especially for you. Secondly, you will be supporting the programmers who put a lot of effort into this package. Your support will ensure a Hacker's Utility Kit V2. The Hacker's Utility Kit is FAIRLY priced at $19.95 from Computer Mart or any dealer who supports the Kracker Jax Line of products.

Super Snapshot 64

What are we doing recommending a snapshot type utility to you? We know you would rather hand break programs if at all possible. What we would like to let you know about is the M/L monitor available in this cartridge. This monitor has proven to be an invaluable tool time and time again. There is no place in memory you can't access. Even screen memory is accessable. All this from within a running or booting program. Let the program boot until you suspect protection check and stop the program cold. Today's protection is being placed in areas of memory that are increasingly difficult to access. This monitor "reveals" all.

Other features in Super Snapshot are: 1541/71/81 Turbo DOS, DOS Wedge commands, reasonably priced upgrade policy, programmed function keys, a resume feature that allows a program to be stopped, examined and restarted in most cases, totally invisible cartridge when disabled, supports multiple drives, screen dumps to printer as well as disk, and a C-128 mode switch available to make the cartridge invisble to the C-64 mode on the C-128. Allows you to use C-128 software without removing the cartrigde.

Super Snapshot is available from Computer Mart as well as many retailers across the country. This is one purchase you won't regret.

Rent-A-Disk

Well, now you own Kracker Jax Vol II and are excited about a specific protection scheme. What! You don't own that particular program. What to do? Well, you could buy it if you were SURE you wanted to use it anyway or you could borrow it from a friend, if you knew someone who had that particular program. OR, there is another simple solution to the problem. Program renting. Our friends at Rent-A-Disk in West Virgnia have consented to have the commercial programs we have covered in this manual in stock and available to you. Rent-A-Disk allows you to rent a title for your examination for a period of two weeks (includes turn around time). During this time you may try the title before you buy. If you find that the title is what you want, you may purchase that title at a very attractive price. Niether Kracker Jax or Rent-A-Disk condones or encourages piracy in any form. We recommend rental software strictly for trial or, educational purposes. We realize you will have access to copiable software by using this manual, but we urge you to let your conscience be your guide. Please, remember that software piracy is a crime. If you rent a piece of software and you like it, BUY IT. Support the programmers who worked so hard to bring that package to you.

For details on Disk Rentals, call or write Rent-A-Disk at Frederick Bldg. #345, Huntington, WV 25701. (304)529-3232 9am-10pm M-TH 9am-5pm Fr-Sa.